Abstract

Testing safety-critical systems, particularly those controlled by Programmable Logic Controllers (PLC), is crucial for ensuring the safe and reliable operation of industrial processes. This thesis addresses the critical need for automated testing of safety-critical PLC systems used in various industrial settings. Despite the significance of testing, current practices rely heavily on manual methods, leading to challenges in scalability and reliability. This work investigates enabling test automation for PLCs to facilitate and assist the current manual testing procedures in the industry. The thesis proposes and evaluates test automation techniques and tools tailored to PLCs, focusing on Function Block Diagram and Structured Text languages commonly used in industry. We systematically compare test automation tools for PLC programs, after which we propose a PLC to Python translation framework called PyLC to facilitate automated test generation. The experiment employing the EARS requirement engineering pattern reveals that while engineers use semi-formal notations in varied ways to create requirements, leading to completeness issues, it confirms the viability of employing EARS requirements for PLC system testing. Subsequently, the proposed automation approaches are fully implemented and evaluated using real-world PLC case studies, comparing their efficiency against manual testing procedures. The findings highlight the feasibility and benefits of automating PLC testing, offering insights into improving development and testing processes through carefully selected automation tools for the CODESYS IDE, a well-known PLC development environment. Additionally, we show that leveraging Python-based automated testing techniques and mutation analysis enhances testing effectiveness. Furthermore, incorporating best practices in requirement engineering, as demonstrated by the EARS approach, contributes to further enhancing testing efficiency and effectiveness in PLC development.

Sammanfattning

Att testa säkerhetskritiska system, särskilt de som styrs av PLC (Programmable Logic Controllers), är avgörande för att säkerställa säker och pålitlig drift av industriella processer. Denna avhandling tar upp det kritiska behovet av automatiserad testning av säkerhetskritiska PLC-system som används i olika industriella miljöer. Trots betydelsen av testning är nuvarande praxis starkt beroende av manuella metoder, vilket leder till utmaningar i skalbarhet och tillförlitlighet. Detta arbete undersöker att möjliggöra testautomatisering för PLC:er för att underlätta och hjälpa de nuvarande manuella testprocedurerna i branschen. Avhandlingen föreslår och utvärderar testautomatiseringstekniker och verktyg skräddarsydda för PLC:er, med fokus på funktionsblockdiagram och strukturerade textspråk som vanligtvis används inom industrin. Vi jämför systematiskt testautomatiseringsverktyg för PLC-program, varefter vi föreslår ett PLC till Python-översättningsramverk kallat PyLC för att underlätta automatiserad testgenerering. Experimentet som använder det tekniska mönstret för EARS-kraven visar att även om ingenjörer använder semiformella notationer på olika sätt för att skapa krav, vilket leder till fullständighetsproblem, bekräftar det att det är lönsamt att använda EARS-krav för PLC-systemtestning. Därefter implementeras och utvärderas de föreslagna automatiseringsmetoderna helt och hållet med PLC-fallstudier i verkliga världen, där deras effektivitet jämförs med manuella testprocedurer. Resultaten belyser genomförbarheten och fördelarna med att automatisera PLC-testning, och erbjuder insikter i att förbättra utvecklings- och testprocesser genom noggrant utvalda automationsverktyg för CODESYS IDE, en välkänd PLC-utvecklingsmiljö. Dessutom visar vi att utnyttjande av Python-baserade automatiserade testtekniker och mutationsanalys förbättrar testningseffektiviteten. Dessutom bidrar införandet av bästa praxis inom kravteknik, vilket demonstreras av EARS-metoden, till att ytterligare för-

bättra testningseffektiviteten och effektiviteten i PLC-utveckling.

To my parents

Acknowledgments

I am grateful to my supervisors Eduard Paul Enoiu, Cristina Seceleanu, and Wasif Afzal for their priceless guidance and ideas in designing the research activities, and for providing constructive feedback, and encouragement throughout the thesis. I would also like to thank them for providing me with their kindest support to resolve all challenges encountered in my studies. I'm still on a journey filled with opportunities to learn and grow, with many exciting new experiences ahead of me. I would also like to thank all my co-authors and collaborators for their contributions and support.

I have been lucky to work on real industrial problems at ABB Marine and Ports AB, Sweden. This was made possible by the support of Filip Sebek. I would also like to thank all my colleagues in the VeriDevOps EU project who allowed me to share my ideas with them and establish international collaborations with different European industrial companies and universities such as Ikerlan, Fagor, Montimage, Softeam, and Abo Akademi.

I would like to pay infinite and endless gratitude to my family and my kind girlfriend Sahar for always believing in me and providing me with the confidence, support, and strength that carried me through so many tough times.

I'm also grateful to all my dear colleagues and fellow PhD students for their support and encouragement, especially Damir Bilic, Aldin Berisa, Edin Jelacic, Muhammad Nouman Zafar, and Iliar Rabet.

The work presented in this thesis has received funding from the European Union's Horizon 2020 research and innovation program under the project 'VeriDevOps'.

Mikael Ebrahimi Salari, Västerås, May, 2024

List of Publications

Papers included in this thesis[1]

Paper A: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu. *"Choosing a Test Automation Framework for Programmable Logic Controllers in CODESYS Development Environment".* Published in the 15th IEEE International Conference on IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2022), The Next Level of Test Automation (NEXTA 2022) [1].

Paper B: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu. *"PyLC: A Framework for Transforming and Validating PLC Software using Python and Pynguin Test Generator".* Published in The 38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023) [2].

Paper C: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu *"An Empirical Investigation of Requirements Engineering and Testing Utilizing EARS Notation in PLC Programs".* Submitted to the Springer Nature Journal's Special issue on Topical Issue on Advances in Combinatorial and Model-based Testing 2023 [3].

Paper D: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Cristina Seceleanu, Wasif Afzal, Filip Sebek. *"Automating Test Generation of Industrial Control Software through a PLC-to-Python Translation Framework and Pynguin".*

[1] The included papers have been reformatted to comply with the thesis layout.

Published in the 30th Asia-Pacific Software Engineering Conference (APSEC 2023), Software Engineering In Practice (SEIP) Track [4].

Publications, not included in this thesis

Paper E: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu *"An Experiment in Requirements Engineering and Testing using EARS Notation for PLC Systems".* Published in IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2023), The Advances in Model Based Testing (A-MOST 2023) [5].

Contents

I

Thesis

Chapter 1

Introduction

Industrial Control Systems (ICS) play a pivotal role in the automation of various industrial processes, enabling efficient and reliable operation in sectors such as manufacturing, energy, transportation, and more [6]. Within the realm of ICS, Programmable Logic Controllers (PLC) have emerged as a vital technology, providing the foundation for controlling and monitoring complex industrial systems. PLC programs, written in specialized languages, define the logic that governs the behaviour of these systems, making them integral to the latter's safe and efficient functioning [7]. PLCs offer numerous advantages that make them indispensable in industrial settings. They provide real-time control capabilities, robustness, and flexibility, allowing for the precise coordination of equipment, monitoring of sensors, and execution of critical operations [8].

PLC's ability to interface with various sensors, actuators, and other devices facilitates seamless integration into existing infrastructure, empowering industries to achieve enhanced productivity, reduced downtime, improved quality control, and increased operational safety [9].

Despite the widespread adoption of PLC and its critical role in industrial processes, the proper testing of PLC programs remains a significant challenge. Traditional testing approaches for software applications are often inadequate for PLC programs due to their unique characteristics, which include real-time operation, deterministic behaviour, close interaction with hardware, safety-critical applications, limited debugging capabilities, and long life cycles. PLC programs interact with physical equipment and are subject to real-time con-

straints, making the consequences of errors or malfunctions potentially severe [10]. However, the complexity of PLC programs, combined with the lack of standardized testing methodologies and tools, has led to a significant gap in the automated testing of PLC programs [11].

The consequences of faulty PLC programs can be devastating, resulting in operational disruptions, financial losses, and even threats to human safety. Ensuring the correctness and reliability of PLC programs is crucial to prevent accidents, minimize downtime, and protect critical infrastructure [12]. Therefore, developing and adopting robust automated testing techniques for PLC programs are essential to identify and rectify potential issues early in the development life cycle, reducing risks and enhancing the overall performance of industrial control systems [13].

The lack of automation in generating test cases for PLCs and their unique characteristics, as well as the wide range of different functional and safety/security requirements for them, plus the necessity of aligning PLC programming with different standards, make PLC testing a super challenging task. To tackle this challenge, this thesis investigates the following using several different real-world industrial case studies: enabling automated testing for PLC programs by identifying a proper test automation tool for PLCs in CODESYS IDE via a systematic approach, proposing a fully automated method for translating a PLC program to Python, and performing a deep examination of using a well-known semi-structured requirement syntax such a EARS in terms of PLC testing.

Programming PLC programs using the IEC 61131-3 standard languages [14] has gained significant popularity in the industrial control systems domain. The IEC 61131-3 standard provides a set of programming languages, including Ladder Diagram (LD), Function Block Diagram (FBD), Structured Text (ST), Instruction List (IL), and Sequential Function Chart (SFC), which offer different approaches for developing control logic. These languages provide a standardized and structured approach to PLC programming, facilitating code reusability, modularity, and maintainability. In this work, we focus on FBD and ST languages because of their popularity in the current industry.

Among the various Integrated Development Environments (IDEs) available for IEC 61131-3 programming, we focus on CODESYS IDE in this work since it has emerged as a widely adopted and powerful platform in industry [15]. CODESYS IDE offers a comprehensive development environment that supports all standard languages, enabling engineers to efficiently design, test,

and debug PLC programs [16]. The popularity of CODESYS can be attributed to its user-friendly interface, extensive library of pre-built function blocks, and compatibility with a wide range of hardware platforms.

The related work of this thesis overviews different state-of-the-art studies that investigate the test automation of safety-critical software, especially PLC programs. The reviewed works span three relevant categories including the efforts conducted towards developing or choosing the right test automation frameworks (e.g., [17], [18], [19], [20], [21]), the contributions towards transforming a PLC program to other programming languages (e.g., [22], [23], [24], [25]), and last but not least, the studies that investigate the application and efficiency of using different requirement notations in testing software artefacts (e.g., [26], [27], [28], [29], [30]).

This thesis studies and contributes to the automation of PLC software testing by including a collection of papers. We start by addressing the non-trivial problem of *choosing the right test automation tool for CODESYS IDE* [1]. Next, we propose *PyLC*, a PLC to Python translation framework that introduces the required mechanisms, rules, and workflows during the translation process [2]. Moreover, PyLC introduces a three-layered translation validation mechanism that ensures the validity of the translated code in Python. As the next contribution, we investigate *the applicability of using EARS* [31] *semi-structured requirement notation for PLC testing* by experimenting [3]. *Automating PyLC translation framework*, is the next contribution of this thesis towards enabling and facilitating the PLC testing process.

Motivation: Several cutting-edge methodologies for automated testing of safety-critical embedded systems have been documented in the literature, including those proposed by Li et al. [32], Enoiu et al. [33], Malekzadeh et al. [34], and Prati et al. [35]. Nonetheless, the implementation of these solutions within industrial settings has been sluggish, primarily attributed to platform-specific and domain-focused methodologies that target specific aspects of system functionality [36]. The scarcity of success stories in industrial settings, along with the absence of practical guidelines and a dearth of empirical, evidence-based studies, further hinder the widespread adoption of automated testing techniques [37]. Analyzing the limitations of the current manual PLC testing process in the industry has led us to identify several remarkable research gaps including (i) the lack of test automation and evaluation tools for PLC IDEs, (ii) the necessity of evaluating the effectiveness of test automation

versus manual testing in fault detection for PLCs, (iii) the limited application of state-of-the-art automated test generation tools to PLCs, (iv) the absence of assessing the impact of human modelling of Natural Language (NL) requirements on PLC system certification, and (v) the existence of the non-trivial problem of choosing the right test automation tool among practitioners [18]. All previously mentioned research gaps motivate us to empirically investigate the use of automated testing techniques in practice to test safety-critical embedded systems.

Summary of the Contributions: The lack of automation for testing PLC programs comes with challenges both on the scientific front and the industrial one. In this thesis, we start our work by investigating how to assist researchers and practitioners in choosing the right Test Automation Framework for one of the most popular IDEs in the PLC industry, CODESYS. This work is an attempt to tackle the *non-trivial problem of choosing the right test automation tool among practitioners* [18]. Addressing this problem systematically encounters several challenges, including (i) identifying the practitioners' point of view regarding both the most discussed test automation tools for CODESYS IDE, and important features of test automation tools while excluding the academia's point of view. (ii) Detecting the academia's point of view regarding the reported most important features of test automation tools. (iii) Industrial evaluation of the identified most important features from both academic and practical points of view. (iv) Detecting the most powerful test automation tools of CODESYS IDE through a systematic comparison of the identified tools based on the detected industry-validated features. (v) Evaluating the applicability and efficiency of the identified tools in real-world circumstances by applying them to different industrial case studies. Overcoming the aforementioned challenges has been done in our work [1] by designing a hybrid methodology that utilises several different techniques chained to each other, including Grey Literature Review (GLR) [18], literature review, industrial validation, Test Automation Frameworks (TAF) selection, and TAFs evaluation via a real-world case study.

Furthermore, as the next step towards enabling automated testing for PLC programs considering the existence of a powerful meta-heuristic testing tool in Python called Pynguin [38] and motivated by addressing *the necessity of proper test automation implementation* and *the limited application of state-of-the-art automated test generation tools to PLCs*, we further propose PyLC [2]. PyLC is a PLC to Python framework that proposes the required translation rules,

translation mechanism, workflow, and a hybrid translation validation mechanism for transforming ST and FBD programs to executable equivalent Python code and validating their translation. The proposed hybrid translation validation mechanism of the PyLC tool leverages three different validation mechanisms including Requirement-based testing, Translation Rules-based testing, and automated search-based testing via Pynguin tool [38]. Implementation of PyLC faced several academic and industrial challenges, such as (vi) semantic mapping between the PLC program and its translated code in Python, (vii) implementing and simulating the non-existing data types and modules of PLC such as *TIME* data type and *TON* block in Python, (viii) implementation of the FBD network in Python and preserving the call order of the functions and blocks, (ix) implementation of cyclic execution feature of the PLC programs in Python, (x) validating the correctness of the translation, and (xi) evaluating the applicability and efficiency of the proposed translation framework in real-world circumstances. All the mentioned concrete challenges were addressed in our work which is a proof-of-concept for PyLC translation framework [2].

Motivated by addressing *the absence of assessing the impact of human modelling of NL requirements on PLC system certification*, *the necessity of evaluating the effectiveness of test automation versus manual testing in fault detection for PLCs*, as the next step towards facilitating automated testing of PLC programs and, to the best of our knowledge, a first academic endeavour, we investigate the applicability and efficiency of using a popular semi-formal requirement syntax called EARS (Easy Approach to Requirements Syntax) [31] that has been proposed by researchers at Rolls-Royce. This investigation [3] is carried out by experimenting with transforming a selected set of security requirements originally expressed in NL into EARS requirements. This work continues with proposing an *EARS-based PLC testing method* for generating test cases for PLC programs based on EARS requirements and evaluating the applicability and efficiency of EARS syntax in the context of PLC programs by applying it to several real-world case studies. This work also briefly compares the efficiency of the proposed semi-automated EARS-based testing mechanism versus the current manual PLC testing state in the industry. This work encountered several challenges, which were all addressed, such as (xii) identifying the recurring EARS patterns when transforming the NL requirements to EARS requirements, (xiii) ambiguity of NL requirements, and (xiv) catching the functional requirements of the PLC program for requirement-based testing.

Attempting to enable test automation for PLC programs and motivated by filling the gap of *the lack of test automation and evaluation tools for PLC IDEs*, as well as *the limited application of state-of-the- art automated test generation tools to PLCs*, we next automate our proposed PyLC (PLC to Python) framework [4] by equipping it with the following automated modules: *XML Analyzer* module, which extracts the required information from the PLC program under translation, *Python Code Generator* module which generates the executable translated code in Python, and finally, the *Meta-heuristic Test Generator* module which validates the correctness of the translation using search-based algorithms. The automated version of the PyLC tool follows the proposed translation rules and translation procedures of the previous manual version [2], but it is capable of importing a PLC program in FBD language (as a PLCopen XML[1] file) and translating it into an executable Python code automatically. The automated PyLC is also capable of validating the translation, by employing the search-based testing algorithms of the Pynguin tool [38]. The automation of PyLC encountered several technical challenges, which were all addressed, such as (xv) proper automated data extraction from the PLC program, (xvi) preserving the FBD network as well as the order of the block execution in the PLC program under automated translation, (xvii) automated careful conversion of data types to equivalent or similar data types in Python, (xviii) simulating the behaviour of the non-existing blocks of PLC (e.g., TON, TOF) in Python automatically.

Results: The comprehensive investigation and evaluation of the proposed PLC testing and requirement engineering methods used in this thesis show that:

- The most prevalent test automation frameworks targeting CODESYS IDE for PLC testing are the CODESYS Test Manager and CoUnit [1].

- Several features that we have identified should be considered when choosing a test automation framework for PLC testing: cost, supported platforms, industrial use, stage of development, documentation and report generation, record playback, test suite support, test suite extension, team support, DevOps/ALM support, continuous integration support, scripting language, import support, availability of customer support, quality of documentation, and maintenance support [1].

[1] https://plcopen.org/technical-activities/xml-exchange

- Based on our initial comparison between CODESYS Test Manager and CoUnit, based on the 15 industry-validated identified features, it follows that CODESYS Test Manager is more mature and has several advantages over CoUnit, including user support, record and playback features, and easy test suite extension. Nevertheless, CoUnit, as an open-source counterpart, also provides testers with many key features used during PLC testing [1].

- Our proposed PyLC framework is capable of translating a PLC program into an executable Python code, and its applicability and efficiency seem promising, based on our evaluation by applying it to different industrial real-world PLC programs [2].

- The hybrid unit testing mechanism of PyLC validates the correctness of the obtained Python code, by achieving 100% coverage for requirements-based testing and translation-rules-based testing methods, and an average branch coverage of 88.44% for the search-based testing method [2].

- The results of the conducted experiment in requirements engineering and testing using the EARS notation for PLC systems imply that different individuals use different EARS patterns for transforming the same requirement, based on their interpretation, which shows an acceptable level of flexibility in the EARS syntax [3].

- The results from the testing part of the conducted EARS notation experiment and the subsequent comparison with traditional PLC testing methods indicate that EARS-generated requirements-based test cases for PLC programs are effective and offer an accessible means for PLC testers to express test specifications [3].

- The automated PyLC framework demonstrates the capability for translating efficiently an array of industrial FBD programs, characterized by diverse block types, into Python code [4].

- The automated 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 [4].

Outline of the Thesis: The thesis consists of two parts. Part I provides an overview of the conducted research and is organized as follows: Chapter 2 provides a brief overview of the background along with related work, Chapter 3 presents the research goals, methodology, and research process, Chapter 4 summarizes the included papers and contributions, Chapter 5 briefly overview the results of this thesis, Chapter 6 deals with discussion and limitations of our proposed approaches, followed by conclusions and future work in Chapter 7. Part II includes the published papers, which have been adapted to comply with the format of the thesis.

Chapter 2

Background & Related Work

This chapter overviews the fundamentals of the development of safety-critical embedded software, the purpose of its testing in the industry, unit testing techniques, requirement-based testing, test coverage, and mutation analysis, followed by related work that has been carried out in similar domains.

2.1 Development and Testing of Safety-Critical Software in Industry

Embedded systems, as outlined by [39], encompass both hardware and software elements interfacing with the physical world via sensors and actuators to influence the environment. These systems are purposefully crafted to streamline the execution of intricate tasks, thereby reducing human effort and time consumption. Their widespread adoption across various sectors, including aviation, transportation, and nuclear power plants, underscores their significance. Failures within these systems, as noted by [40], can result in detrimental impacts on human life, the environment, and economic stability. The initiation of safety-critical software development within the industry commences with requirement analysis, employing qualitative and quantitative techniques such as fault tree analysis, expert analysis, etc. [41]. Following the requirements analysis phase, the implementation of safety-critical software typically occurs on specialized computers referred to as Programmable Logic Controllers (PLCs).

These PLCs are tasked with executing the safety-critical functions of a system. They receive input signals from sensors, execute computational logic, and transmit instructions via a computer network to various modules and subsystems for executing safety-specific tasks.

2.1.1 IEC 61131-3 Standard and PLC Programming

The International Electrotechnical Commission (IEC) has established multiple programming languages for implementing safety-critical software applications on PLCs, including Instruction List (IL), Structured Text (ST), Function Block Diagram (FBD), among others [42]. Within this framework, FBD and ST stand out as two of the most popular PLC graphical and text-based programming languages in the industry, respectively. Consequently, this thesis is focused on PLC programs developed in FBD and ST languages.

Structured Text (ST)

ST offers a structured and intuitive approach to developing control algorithms for industrial automation systems. ST enables engineers to express complex logic and algorithms using familiar constructs such as sequential, selection, and iteration statements, akin to high-level programming languages. This facilitates the design and implementation of sophisticated control strategies for various industrial processes. Additionally, the readability and maintainability of code are enhanced through the use of structured programming techniques, promoting better understanding and easier troubleshooting by technicians and programmers alike. Moreover, ST's standardized syntax and semantics across different PLC platforms contribute to the interoperability and portability of control software, allowing for seamless integration and scalability in diverse industrial environments. As such, structured text programming language serves as a cornerstone in the development of reliable, efficient, and flexible control systems for industrial automation applications.

Function Block Diagram (FBD)

FBD employs a graphical modelling notation to depict various functions and *function blocks*, such as arithmetic operations, selection processes, comparisons, and more. These function blocks are interconnected via input and output

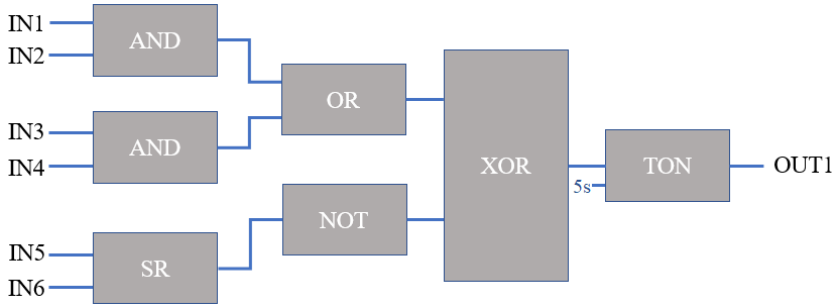

Figure 2.1: An example of FBD program with six inputs and one output

variables, delineating the functional properties and interrelationships among different components of the software application, as defined by the functional and non-functional requirements, respectively. Figure 2.1 illustrates an instance of an FBD program comprising arithmetic operators (AND, OR, NOT, XOR), a latch (SR), and a timer (TON) function block. This program takes six parameters/signals as inputs and yields a single output based on the logic depicted by functional blocks, with a delay of five seconds. Subsequently, the FBD programs are compiled using specialized industrial compiling tools, thereby converting them into source and machine code.

2.1.2 Testing of PLC Safety-Critical Software in Industry

Verification and validation of safety-critical software constitute an iterative process conducted throughout the development lifecycle to ensure its behavioural functionality aligns with system requirements. In industrial settings, the testing process commences concurrently with software development, adhering to the V-model, which enjoys widespread acceptance among practitioners in embedded software development [43], [44], [45]. However, various iterations of V-models [46] are employed across different industries based on their specific business requirements. In this section, we offer a simplified depiction of the V-model, as illustrated in Figure 2.2, to elucidate the foundational concepts utilized in this thesis.

The V-Model comprises four distinct phases: *the development phase, the test design phase, the testing phase*, and *the implementation phase*. Figure 2.2

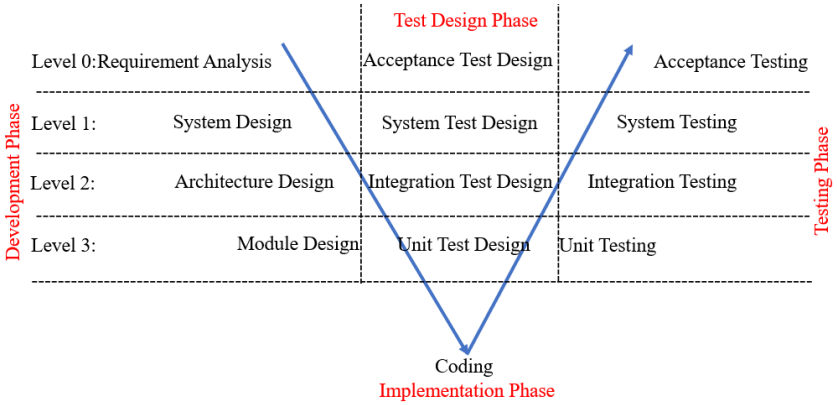

Figure 2.2: A simplified version of a V-model

depicts the development phase on the left side, detailing each activity of the development life cycle at an abstract level as described by Shuping et al. [45]. The functional specification's depth increases at each level: Level 0 focuses on the customer perspective, Level 1 delineates the functional design of the entire system, Level 2 specifies data transfer and communication details between modules and the external environment, and Level 3 provides detailed functional specifications at the unit level for each module. The test design phase runs concurrently with the development phase, generating test specifications for each level. The implementation phase involves the actual development of system modules to fulfil requirements. Subsequently, during the testing phase depicted on the right side of the figure, test designs are executed iteratively and incrementally following the development of each module.

The test execution phase within the embedded system industry typically unfolds across three tiers, namely Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), and Hardware-in-the-Loop (HiL) [39]. During the MiL phase, tests are conducted on a model that represents the system requirements, aiming to validate both the model's compliance with requirements and its computational logic. In contrast, the SiL phase involves running tests on the real software alongside experimental hardware, simulating the behaviour of actual hardware. Conversely, the HiL phase entails conducting tests on both real software and hardware within a simulated or virtual test environment.

The testing complexity and effort required for individual activities within the testing phase escalate incrementally [44]. Each distinct activity encompasses varied test objectives, input/output parameters, and communication formats. At the unit and integration levels, the test objectives and input/output parameters are confined to the functionality of smaller system components. Conversely, complexity at the system level surges alongside the expansion of the input/output parameters and the integration of additional modules, thereby augmenting the testing effort at the system level.

2.2 Unit Testing Techniques

The fundamental objective of unit-testing techniques is to produce precise and robust test artefacts, including test cases and scripts, to verify system requirements and ensure their reliability. Various automated testing methodologies are available, such as mutation testing [47], boundary testing [48], equivalence partitioning [49], and code coverage analysis [50], which can be employed to automatically generate test artefacts for systematically validating a System Under Test (SUT), thereby minimizing costs in terms of time and effort. Nonetheless, manual testing remains a widely utilized approach in the industry, often complementing automated software testing [51]. These techniques are adaptable across different testing levels, including component and integration testing. However, in this study, search-based testing, mutation testing, and code coverage analysis are executed at the unit level to assess the system's adherence to its prescribed requirements empirically. Several studies, such as those by Hametner et al [52], Jamro et al. [19], Winkler et al. [53], Li et al. [54], Dhadyalla et al. [55], and Rengarajan et al. [56], have demonstrated the efficacy and efficiency of these techniques in validating safety-critical software at the unit level. This thesis is focused on the automated testing of PLC programs at the unit level.

2.2.1 Manual Testing

During manual testing of a safety-critical system, test cases are manually crafted in accordance with established safety standards outlined by various organizations (e.g., ISO [57], IEC [58]). This process involves leveraging requirements and test specifications to formulate test cases aligned with

specific test objectives and various structural or behavioural coverage criteria, such as statement coverage and input space partitioning, respectively. The test cases are articulated in natural language, delineating test steps comprising input, expected output, and constraints based on system requirements. Subsequently, these test cases are transformed into tangible test cases or test scripts, which can be executed either manually or automatically on the System Under Test (SUT) to generate test verdicts. This thesis attempts to assist the current manual PLC testing process by enabling the scientifically proven state-of-the-art testing mechanisms for PLC programs [2], [4].

2.2.2 Search-based Testing

Search-Based Testing (SBT) is a systematic approach that utilizes search algorithms to automatically generate test cases to achieve specific testing objectives. This method has earned significant attention in the testing community due to its effectiveness in exploring complex search spaces and identifying diverse test scenarios [59]. In the context of safety-critical systems, where thorough testing is paramount, SBT offers several advantages. By leveraging various search strategies, such as genetic algorithms or simulated annealing, SBT can efficiently navigate through the extensive input space of safety-critical systems, thereby increasing the likelihood of uncovering critical faults or vulnerabilities. Furthermore, SBT can be tailored to target specific safety properties or requirements, enabling testers to focus their efforts on areas of particular concern. Research by Querejeta et al. [60] and Doganay et al. [61] highlight the successful application of SBT in testing safety-critical systems, demonstrating its potential to enhance testing practices and contribute to the overall reliability and robustness of such systems. This thesis allows PLCs to benefit from different supported search-based testing algorithms of the integrated testing tool inside the proposed PLC to Python translation framework [2], [4].

2.2.3 Requirement-based Testing

Requirement-based testing is a cornerstone in the development and verification of safety-critical systems, ensuring that every aspect of the system's functionality is thoroughly examined against its specified requirements [62]. By adhering closely to the requirements documentation, testers can systematically

derive test cases that cover various scenarios and use cases, including normal operations, edge cases, and failure modes. This meticulous approach not only validates the system's compliance with safety standards but also helps uncover potential design flaws, implementation errors, and operational risks that could compromise safety. Moreover, requirement-based testing facilitates traceability, enabling stakeholders to trace test cases back to specific requirements, thereby fostering transparency and accountability throughout the development process. In the realm of safety-critical systems, where even minor errors can have catastrophic consequences, the rigorous application of requirement-based testing methodologies, guided by established standards like IEC 61508 [58] for safety-related systems, and IEC 61131-3 [14] for PLCs, plays a pivotal role in mitigating risks and ensuring the utmost safety and reliability of the system. Inspire by the popularity of requirement-based testing in the current testing procedure of PLCs in industry, this thesis investigates automated PLC testing using both functional and safety-related requirements of different real-world use cases [1], [2], [3], [4].

2.3 Test Coverage

The metric used to assess the comprehensiveness of a test suite in software testing is known as *test coverage*, which measures the extent to which elements such as code and requirements are covered by the suite at either the design or implementation level [63]. Various criteria, including branch coverage, statement coverage, and requirement coverage, are employed to assess and produce test suites. However, this thesis conducts an empirical comparative assessment of test suites generated using *requirement coverage and branch coverage*. Requirement coverage and branch coverage are chosen due to their status as the de facto standard in test suite creation for many industrial control system manufacturers, including our industrial partner (e.g., ABB Marine and Ports). Furthermore, various investigations (e.g., [64], [65], [66], [67]) have provided evidence of the efficacy of requirement-based test suites and branch coverage in validating safety-critical systems.

2.3.1 Requirement Coverage

In the context of a safety-critical system, evaluating whether every safety and domain requirement outlined in the requirement specification is addressed by a test suite at minimum once [68] is crucial. Specifying the requirements in natural language by a pattern-based syntax such as EARS [31] can facilitate the test generation from the requirements by structuring the NL requirements via different provided patterns.

Requirement coverage stands as a fundamental black-box coverage criterion utilized to evaluate the behavioural coverage of a test suite. The requirement coverage is an easy-to-understand criterion which is popular in current existing testing procedures in the industry. This criterion quantifies the overall number of implemented requirements and identifies any undocumented requirements in the implementation, alongside determining the total count of test cases necessary to cover each requirement.

2.3.2 Branch Coverage

Branch coverage, a metric in software testing, measures the proportion of branches executed during test execution compared to all possible branches in the code. It is a critical aspect of assessing the thoroughness of test suites, ensuring that various execution paths within the code are exercised. Different studies highlight the importance of branch coverage in identifying potential defects and enhancing the reliability of software systems [69], [70], [71]. For instance, achieving high branch coverage in safety-critical domains like aerospace and medical devices is imperative to mitigate risks associated with undetected faults [72]. Moreover, other studies emphasize the correlation between branch coverage and fault detection effectiveness, indicating that higher branch coverage often leads to improved fault detection rates [73]. Therefore, by focusing on branch coverage during testing, developers can enhance software quality, reduce the likelihood of system failures, and ultimately deliver more reliable software products. In different parts of this thesis, we employ branch coverage in evaluating the quality of the conducted PLC testing under the IEC61131-3 standard [1], [2], [3], [4].

2.4 Mutation Analysis

Test coverage criteria quantify the extent of code exercised by a test suite, offering insights into potential enhancements for test adequacy. However, despite achieving high coverage, studies [74], [75] indicate that various factors can impede the fault detection efficacy of a test suite.

In this thesis, we carry out a comprehensive evaluation approach that includes assessing the fault detection effectiveness of generated test suites through mutation analysis [76]. Mutation analysis is known as one of the most valid test evaluation techniques in academia and involves creating mutated versions of the original program and introducing small faults representing common programming errors or logical flaws. Mutants are classified as equivalent or non-equivalent; the former maintains behaviour akin to the original program, while the latter exhibits divergent behaviour. Test suites, designed using specific techniques, are then executed on both the original and mutated versions. A mutant is considered killed if test results differ between the original and mutant versions, indicating effective fault detection. The mutation score, computed based on strong and weak mutation criteria, reflects the number of killed mutants, providing a robust evaluation of test suite effectiveness. This thesis enables automated mutation analysis for PLC testing via translating them to executable Python code and reporting the mutant coverage in different publications [2], [4].

2.5 Related Work

This section deals with state-of-the-art studies that have explored and investigated the use of test automation in PLC testing. The related work to this thesis can be divided into three main categories:

- Studies that assess the effort of researchers towards developing or choosing the right test automation frameworks (e.g., [17], [18], [19], [20], [21]).

- Contributions towards transforming a PLC program to other programming languages (e.g., [22], [23], [24], [25], [77]).

- Studies that review the application and efficiency of using different requirement notations in testing (e.g., [26], [27], [28], [29], [30]).

- Studies related to testing embedded industrial systems (e.g., [78], [79], [80], [81]).

2.5.1 Developing or Choosing The Right Test Automation Frameworks

In recent years, researchers have made efforts to develop test automation frameworks for PLC software. Jamro introduces a method for POU-oriented unit testing for IEC 61131-3 languages [19]. In this approach, test cases are defined in CPTest+, a dedicated test definition language. The proposed approach is introduced in the CPDev engineering environment. Recently, Hofer and Russo [20] presented a unit-testing framework named APTest (Advanced Program Organization Unit Testing) for CODESYS IDE. The framework is developed based on the IEC61131-3 standard and CPTest+. APTest is a POU-based framework equipped with a test library supporting different types of assertions and is compatible with CODESYS (version 2.3).

Even if these academic tools have a wide range of capabilities, such as test parallelization, simulating analogue signals, and supporting time-dependent behaviours, there is limited evidence of how industrially useful these frameworks are. In addition, these tools are only compatible with older versions of CODESYS. Selecting a test automation framework is an essential part of software testing, and recent studies have looked at different challenges to implementing automation support. Raulamo-Jurvane et al. [18] performed a GLR to identify the practitioners' criteria for choosing the right test automation tools. The study showed that practitioners select and embrace the widely known and utilized tools.

Garousi et al. [17] compared visual GUI testing frameworks (i.e., Sikuli and JAutomate) using several relevant features and performed an industrial case study. In 2019, Raulamo-Jurvane et al. investigated the practitioners' opinions on evaluating testing tools by conducting an online survey [21]. They found that evaluations in which one uses a tool seem to be more favourable than those based on opinions, and considering the opinions of seven experts provides a reasonable level of reliability.

These results kindled our interest in studying how to tackle the problem of choosing a test automation framework for PLCs in CODESYS, especially when these tools are used to test safety-critical industrial control systems. Motivated by this, this thesis extends the previous efforts of assisting the selection of a proper testing tool by addressing the non-trivial problem of choosing the right automation testing tool for one of the most popular PLC IDEs using a hybrid methodology consisting of GLR, literature review, case study, and a systematic comparative study [1].

2.5.2 Transforming a PLC Program to Other Programming Languages

Marcel et al. [82] 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 [24]. 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. [23] proposed a toolbox that can formalize logic coverage criteria and use it inside a model-checker to generate test cases [23]. 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 to transform 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. [77] 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 data-flow 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.

Previous contributions in transforming PLC programs to other languages range from SCs-based approaches (e.g., [82]) and the ones using the C language (e.g., [77]) to model-based approaches of transforming the actual FBD program code (e.g., [23]). The technique in [58] is based on the IEC 6150 models and supports other parts of the development process. However, compared to our work, these works do not cope 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.

In the context of IEC 61508 standard [58], Mirko Conrad [22] 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. Technically speaking, this work utilises manual numerical model-based equivalence testing to identify the absence of unintended functionality in the context of the IEC 61508 standard, whereas the PLC to Python translation contribution of this thesis is automated and is focused on requirement-based testing of PLC programs at both unit and system levels in the context of the IEC61131-3 standard.

2.5.3 Application and Efficiency of Using Different Requirement Notations in Testing

Mavin and Wilkinson [26] reflected on the ten years of EARS [31] and shared some lessons learned in their review paper. For example, they discovered that EARS users manage to author more useful draft requirements as they incrementally work to find the appropriate EARS pattern. They recommend that new engineers write several requirements and seek expert review with the application of EARS being more useful if one can apply the following activities:

training, thinking, semantics, syntax, and review. In our study, we confirm some of these results even if we do not cover all of the activities stated.

Mavin et al. [27] report on the understanding of four experienced EARS practitioners and their reflections on their experiences of applying EARS in different projects and domains over six years. They report the following EARS-specific lessons learned: training should be short, use EARS with or without a tool, use coaching to embed learning, challenge the EARS Patterns, and question if the EARS clauses are necessary and sufficient.

Mäntylä et al. [28] performed a controlled experiment on test case development and requirement review and the effects of time pressure. They saw no statistically significant evidence that time pressure would lower effectiveness or provoke negative influences on motivation, frustration, or performance.

Dalpiaz et al. [29] investigated the adequateness, completeness, and correctness of use cases and user stories for the manual creation of a static conceptual model. They performed a controlled experiment with 118 subjects, and their results show that user stories work better than use cases when creating conceptual models. Furthermore, user story repetitions and conciseness contribute to these results. However, as we aim with our study, more evidence needs to be provided regarding the aspects that must be considered when selecting and using a modelling and requirement notation.

Weninger et al. [30] report the results of a controlled experiment in which they compared two approaches for defining restricted use case requirements from multiple perspectives, including misuse, understandability, and restrictiveness. Their results indicate the usefulness of the restricted use case modelling approach.

To the best of our knowledge, at the time of writing this thesis, the applicability and efficiency of using a scientifically proven semi-formal requirement notation such as EARS for testing PLC programs has not been investigated by other researchers. This identified research gap leads us to investigate the applicability of EARS requirement notation for PLC programs under the IEC61131-3 standard and propose an *NL requirement to PLC testing mechanism* which is applied to different several real-world case studies [3].

2.5.4 Testing Embedded Industrial Systems

Jee et al. [78] presented an automated test case generation approach for FBD programs, utilizing chosen test coverage criteria to generate test requirements. By employing an SMT solver, the method effectively generates test cases that meet the desired coverage goals. A case study on reactor protection systems demonstrates the effectiveness of the approach in detecting real errors and mutants, outperforming manual test suites prepared by domain experts. The study suggests that automated test case generation for complex FBD programs is both feasible and highly efficient, with the FBDTester tool offering assurance to test engineers working on safety-critical software. This work differs from our related efforts in the context of automated testing for PLC programs [2], [4] in terms of modelling the functional requirements and the type of test generation algorithms used. Moreover, this work is focused on only FBD programs while the initial version of our proposed translation framework enables automated testing for PLC programs in both ST and FBD languages [2].

In a similar effort, He et al. [79] introduced STAutoTester, a framework for automatically generating test cases for ST programs used in PLCs. Leveraging Dynamic Symbol Execution (DSE) and redundant path pruning, STAutoTester efficiently generates test data under various coverage criteria. Evaluation of 21 programs demonstrates its effectiveness, achieving comparable statement coverage with fewer test cases than previous symbolic execution-based tools. The framework supports both structural and logical coverage criteria and shows potential for enhancing automated testing efficiency for PLC software. Evaluation of performance under different path search strategies and extending the coverage to data flow testing are not covered in their work. This work differs from our work in terms of both the supported IEC 61131-3 languages and the test case generation algorithms.

Dobslaw et al. [80] proposed the MC-TOA framework, which offers efficient test set selection for large-scale industrial systems, accommodating diverse search criteria. Compared to state-of-the-art methods like Borg and random search, MC-TOA demonstrates superior performance and versatility in real-world applications. It enables fast multi-objective optimization, providing valuable insights into industry-relevant metrics and bridging the gap between research and industry needs. Despite the valuable contributions of the work towards test set selection for large-scale systems in the industry, exploring dy-

namic search, formulation complexity, scalability, and reinforcement learning techniques to further enhance test set optimization are not discussed in this paper. This work differs from our work in terms of the level of testing. Moreover, the goal of this work is to assist in the efficient test set selection, whereas the goal of our similar work is to enable and facilitate the test automation process for PLCs.

Gargantini et al. [81] present a model-driven environment for hardware/-software co-design and analysis of embedded systems, leveraging UML profiles for SystemC/multithread C and the Abstract State Machine (ASM) formal method. It introduces a methodology based on UML 2, SystemC, and ASM, facilitating graphical representation, code generation, and system validation. The work aims to address the lack of formal analysis techniques in system-level design, proposing a solution that integrates UML-based modelling and ASM formalism. Key components include the ASMETA toolset for ASM modelling and analysis. The paper also discusses the environment's architecture, highlighting its support for high-level functional validation and conformance testing. This effort differs from our work in terms of the layer of testing and the type of embedded systems.

To the best of our knowledge, to the moment of writing this thesis, in the context of automated PLC testing, there has been no deep investigation towards enabling automatic search-based testing via different meta-heuristic algorithms of a powerful Python test generator called Pynguin [38]. This has been done through different included publications of this thesis [2], [4].

Chapter 3

Research Overview

This chapter provides a brief description of the research goals along with the methodology used to conduct the research activities to achieve the defined research goals.

3.1 Motivation & Research Goal

Testing PLC programs in today's industrial control systems has always been a crucial task for industrial automation companies all around the world. Considering the wide application range of PLC programs in the world of embedded systems such as nuclear plants and cranes, proper testing of PLC programs can save human lives as well as the time and energy of automation companies. The current test generation for PLCs is done manually in industry [83], which demands experienced testers. Despite its benefits, this manual testing procedure is time and energy-consuming for companies and is exposed to human errors. Moreover, due to the Industry 4.0 revolution [84], today's PLC programs are getting larger and more complex than before, which makes them even harder to test. The current context and the future landscape of PLC testing in the industry demand increased attention from both academics and practitioners. Motivated by finding a proper and efficient solution to this problem, in this section, we analyze the current limitations of manual PLC testing to identify the existing Research Gaps (RGp) in this context. We identify the following as some of

the most important existing RGp in the current manual unit testing of PLC programs:

- **RGp**$_1$: The current Integrated Development Environments (IDEs) for PLC lack automation of test generation and test evaluation (e.g., by mutation testing). This can lead to multiple technical and scientific challenges such as:
  - Limited Test Suitability: Without automated test generation and evaluation features, developers may struggle to create comprehensive test suites that adequately cover all aspects of PLC functionality, increasing the risk of undetected faults.
  - Increased Development Time: Manual creation and evaluation of tests can significantly lengthen the development cycle for PLC-based projects, delaying time-to-market and hindering project deadlines.
  - Difficulty in Test Maintenance: Manual testing processes are prone to errors and inconsistencies, making it challenging to maintain and update test suites as PLC systems evolve or requirements change.
  - Lack of Traceability: Without automated test generation and evaluation tools, it can be difficult to trace test results back to specific requirements or code changes, impeding the debugging and troubleshooting process.
  - Reduced Confidence in Testing: Manual testing methods may lack the rigour and repeatability of automated approaches, leading to uncertainty about the reliability and effectiveness of test results.
  - Risk of Human Error: Manual test generation and evaluation are susceptible to human error, potentially overlooking critical test scenarios or introducing biases that skew the testing outcomes.
- **RGp**$_2$: If not implemented properly, test automation applied to test creation will be less effective than manual testing in detecting faults [85], [86], [87]. This can lead to several scientific challenges, such as compromising the reliability of test results, inhibiting comprehensive fault coverage, and impeding accurate assessment of system performance and robustness. Additionally, it may hinder the identification and resolution

of potential issues early in the development process, ultimately prolonging the time-to-market for critical systems.

- **RGp**$_3$: The limited application of state-of-the-art automated test generation tools to PLC and corresponding development environments can impact the use of test automation for test creation in industrial practice. This can lead to serious challenges such as:
  - Limited Test Coverage: Insufficient automation may result in incomplete test coverage, leaving potential faults undetected, thereby compromising the reliability of PLC-based systems.
  - Difficulty in Scalability: Manual testing approaches often struggle to scale effectively with complex PLC systems, hindering the ability to adequately assess large-scale industrial setups.
  - Resource Intensiveness: Manual testing consumes substantial human resources and time, which could otherwise be allocated to more strategic tasks, affecting overall productivity and efficiency.
  - Maintainability Issues: Manual testing procedures may become increasingly difficult to maintain and update as PLC systems evolve or undergo modifications, leading to inconsistencies in testing practices.
  - Reduced Agility: Manual testing can impede the agility of development cycles, slowing down the pace of innovation and adaptation to changing industrial requirements.
  - Validation Challenges: Inadequate automation may pose challenges in validating PLC systems against stringent industrial standards and regulations, potentially leading to compliance issues and safety concerns.
- **RGp**$_4$: Engineering PLC systems commonly demand certification according to safety standards that impose specific constraints on requirements engineering and specification-based testing. Since requirements are often expressed in natural language, there is little evidence of the extent to which humans can effectively model requirements and how the modelling impacts the development and testing of PLC systems. This gap can generate several technical and scientific challenges, such as:

  - Ambiguity in Requirements Interpretation: Natural language requirements can be prone to ambiguity, resulting in misinterpretations during the modelling process, which may lead to inconsistencies and errors in PLC system development and testing.
  - Lack of Formalization: The absence of formalized requirements modelling techniques can hinder the systematic translation of requirements into testable specifications, complicating the verification and validation processes for PLC systems.
  - Difficulty in Requirement Traceability: Without structured requirements models, tracing individual requirements throughout the development lifecycle becomes challenging, impeding the ability to ensure that all functional and safety-critical aspects are adequately addressed.
  - Complexity in Verification: Human-modeled requirements may introduce complexity in the verification of PLC systems, making it difficult to ascertain whether the implemented system accurately reflects the intended functionality outlined in the requirements.
  - Risk of Incomplete Coverage: Incomplete or inaccurately modelled requirements may result in gaps in test coverage, leaving potential hazards and faults undetected, thereby jeopardizing the safety and reliability of PLC-based systems.
  - Compliance and Certification Hurdles: Insufficiently modelled requirements may lead to difficulties in satisfying regulatory compliance and certification requirements, delaying the deployment of PLC systems in safety-critical industrial environments.

- **RGp**$_5$: Selecting the right test automation tool is a non-trivial task for many practitioners [18]. This could stem from at least two reasons: (i) not knowing what criteria are important to use for choosing the right tool, and (ii) the lack of knowledge of the pros and cons of using particular test automation frameworks in practice. This can lead to multiple challenges, such as:

  - Suboptimal Tool Selection: Without a clear understanding of selection criteria and the strengths and weaknesses of various automation frameworks, practitioners may inadvertently choose tools that

are ill-suited for their specific testing needs, resulting in suboptimal outcomes.

- Ineffective Test Automation: Inadequate knowledge of test automation frameworks may lead to their improper utilization, resulting in ineffective test automation strategies that fail to achieve desired levels of efficiency and fault detection.
- Limited Innovation: The absence of informed decision-making in selecting automation tools may hinder innovation in test automation practices, preventing practitioners from leveraging cutting-edge technologies and methodologies to improve testing processes.
- Wasted Resources: Misguided tool selection may result in wasted resources, as practitioners invest time and effort into implementing automation solutions that ultimately prove unsuitable or inefficient for their requirements.
- Reduced Competitiveness: Inability to select appropriate test automation tools may lead to decreased competitiveness in the market, as competitors who employ more effective automation strategies gain an advantage in terms of product quality, time-to-market, and cost-effectiveness.

To provide a clearer picture of how test automation can be applied for different parts of the testing process in PLC, we include an overview of automation across the software testing process in Figure 3.1, which is proposed by Garousi et al. in [88]. As observed in the figure, a software testing process consists of five main stages (marked with green boxes) including *Test-case Design*, *Test Scripting*, *Test Execution*, *Test Evaluation*, and finally, *Test-result Reporting* respectively. Each of these main steps can be done either manually (M), using automated tools (A), or by mixing the two (A/M).

Enabling test automation for PLC, as the main goal of this thesis, can benefit all these five stages of software testing as follows: (i) Enabling semi-structured requirement notation and automated test case generation for PLC programs can assist "Test-case Design" and "Test-scripting" phases, (ii) Enabling powerful test evaluation mechanisms such as mutation analysis for PLC can assist the "Test Evaluation" step in the PLC testing process, (iii) Guiding

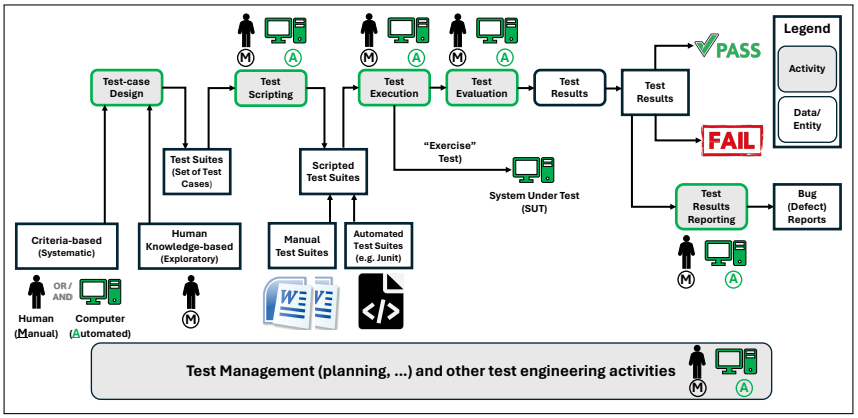

Figure 3.1: An overview of automation across the software testing process proposed by Garousi et al. in [88]

.

practitioners to choose the right test automation tools for PLC programs can assist the remaining testing stages, that is, "Test Execution" and "Test-result Reporting". It is worth mentioning that all research gaps above are addressed in this thesis via different publications [1], [2], [3], [4].

Overall Research Goal. This subsection presents the main goal of the thesis, which is divided into smaller sub-goals, to address the problem thoroughly and analyze it in more detail. Given the previously described problem, the overall Research Goal (RG) of this thesis is:

- **RG**: To facilitate and evaluate test automation throughout the unit and integration testing processes of industrial PLC programs.

Aiming to achieve the overall research goal of this thesis, we formulate the following research sub-goals (R-SG), which act as the main ingredients of the main goal:

- **R-SG$_1$**: Facilitate the use of test automation for the generation and execution of test cases for PLC programs.

  Achieving R-SG$_1$ is an essential step towards acquiring the main goal of this study because meeting "test automation" for PLC programs written

in two popular languages including FBD and ST, starts with generating test cases and executing them automatically. This can be preferably done by using an already existing Python-based automated test case generation tool that uses testing algorithms deemed efficient. A challenge to achieve this is bridging the possible gaps between the two worlds of PLC and Python.

This R-SG investigates two main directions, including the systematic selection of a test automation tool for PLC programs and the development of a PLC to Python transformation that facilitates the use of search-based unit tests in the context of PLC testing.

- **R-SG$_2$**: Investigate the use of semi-formal requirements for engineering and test automation of industrial PLC programs.

  Meeting R-SG$_2$ is a crucial step towards achieving the main goal of this thesis since using an already existing scientifically-proven requirement engineering notation can facilitate the requirement-based testing, therefore unambiguous requirements could serve test generation automation.

  This R-SG relates to research on the applicability and efficiency of using semi-formal requirement patterns in terms of engineering and testing PLC programs.

- **R-SG$_3$**: Evaluate the applicability, efficiency and effectiveness of the proposed PLC test automation approaches in an industrial context.

  A practical evaluation of the applicability and efficiency of the proposed PLC testing approaches using real-world industrial use cases is a remarkable step towards achieving the main goal of this thesis since it investigates the usefulness of the academically developed tool in terms of real-world industrial circumstances.

In the following section, we describe the leveraged research method of this thesis by briefly reviewing the research process and mapping the identified research gaps to the contributions of this thesis.

3.2 Research Method

In the field of software engineering, various methods, such as case studies, experiments, and surveys, are employed to conduct empirical research. The choice of method depends on the research objectives and the type of analysis and data interpretation, including qualitative and quantitative approaches [89] [90]. This thesis emphasises the utilization of both qualitative and quantitative data to bring a comprehensive interpretation of research findings for the benefit of the research community and industrial practitioners. To align with our research goals, as outlined in Section 3.1, and data analysis, we have employed the *case study*, *experimentation*, and *literature review* as research methods, as elaborated in the following.

Case studies and experimentation are the chosen research methods for assessing the effectiveness of the proposed solutions through real-world industrial examples. Case studies and experimentation research methods are highly suitable for the task, particularly in the context of PLC software testing [91]. These research methods provide valuable insights into the practical applicability and performance of our proposed solution in real-world settings, allowing for a comprehensive evaluation of its benefits and limitations.

Case studies enable researchers to investigate and analyze specific real-world scenarios, allowing for an in-depth examination of the proposed solution's effectiveness within a specific context [92]. By utilizing real-world industrial examples, researchers can gather rich and detailed data, including user experiences, challenges faced, and outcomes achieved [91]. This qualitative approach enhances the understanding of the proposed solution's practical implications and provides valuable insights into its feasibility, usability, and impact in industrial environments.

Experimentation research methods, on the other hand, allow for a more controlled evaluation of the proposed solution's performance and effectiveness [93]. By designing and conducting controlled experiments, researchers can systematically measure and compare the solution's performance metrics, such as testing efficiency, coverage, reliability, and overall effectiveness [94]. This quantitative approach enables researchers to gather empirical evidence and statistically analyze the results, providing objective insights into the solution's performance and its potential benefits over alternative approaches.

The utilization of real-world industrial examples in both case studies and

experimentation research methods adds significant value to the assessment of the proposed solution. It provides researchers with the opportunity to evaluate the solution's performance under realistic conditions, considering the complexities and constraints typically encountered in industrial PLC software testing scenarios [92]. The use of real-world examples enhances the external validity of the research findings and ensures the relevance and generalizability of the results to real industrial settings. The mapping of research methods with research goals is shown in Table 3.1.

Table 3.1: Research method mapping with the type of data and research goals

Research Goal	Type of Data	Literature Review	Case Study	Experiment	Solution
Subgoal 1	Qualitative & Quantitative	✓	✓	✓	✓
Subgoal 2	Qualitative		✓	✓	✓
Subgoal 3	Qualitative & Quantitative		✓		

3.2.1 Research Process

In our research, we have defined the research process in six iterative steps: (1) Review of industrial systems and processes, (2) Problem identification and formulation, (3) Proposal of a solution, (4) Solution/tool implementation, (5) Validation, (6) Publication of research results and producing the software in Github repository. Figure 3.2 depicts an overview of the research process that we have used in this thesis. This research process aligns with the one commonly followed in academic research. It encompasses key stages such as literature review, problem identification, proposing a solution, implementing the solution, validating the results, and publishing the research outcomes. Even though one cannot find a particular publication that presents various research studies and textbooks discuss these steps individually as part of the overall research process [95], [94].

1. Review of Industrial Safety-critical Systems and Processes: To gain an understanding of PLC systems, a widely used industrial controller, we undertake a thorough investigation. Specifically, we scrutinize the practices and processes employed by industrial practitioners at the ABB ports and Marine automation company in Sweden as one of our industrial partners in the

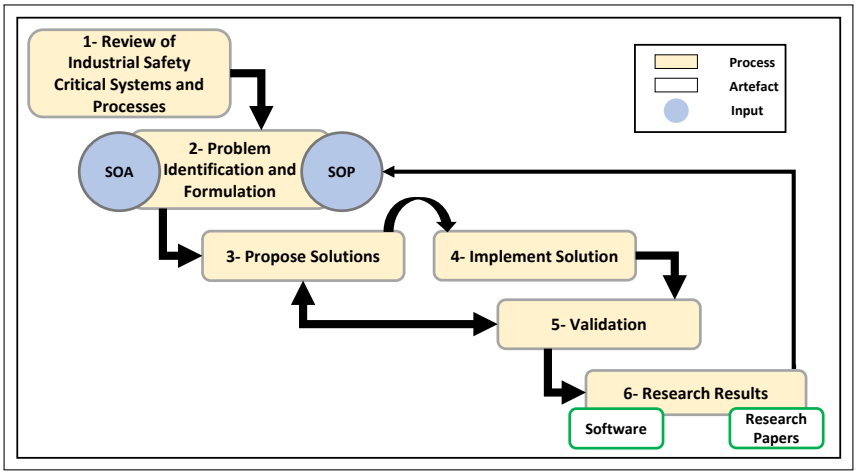

Figure 3.2: Overview of the research process applied in the thesis

VeriDevOps EU project[1]. Additionally, this close collaboration examines the described practices and processes, aiming to identify their requirements and challenges.

2. Problem Identification and Formulation: This step targets the identification of industrial problems based on the analysis carried out in the previous step. We have also formulated the overall research goal (RG) based on the detected research gaps and challenges that exist in the test automation of PLC programs from both State-of-the-Art (SOA) and State-of-the-Practice (SOP) points of view. To gain a clearer understanding of the industrial problem and iteratively address specific aspects, we subdivide the overall goal into several sub-goals. This approach allows us to focus on distinct areas of the problem and investigate them systematically.

Motivated by addressing research gaps RGp1, RGp3, and RGp5, we formulate R-SG1 and R-SG3. In other words, we identify the industrial needs in terms of choosing the right test automation tool and also using automated test generation techniques for PLC programs. Our observations reveal that the current PLC testing procedures in the industry do not incorporate automated

[1]https://cordis.europa.eu/project/id/957212

Table 3.2: Mapping of research sub-goals (R-SG) w.r.t. their respective connection to identified Research Gaps (RGp)

	RGp1 (Lack of test automation in PLC IDEs)	RGp2 (Improper automated test creation)	RGp3 (Limited application of SoA test generation tools)	RGp4 (Ambiguity of test specifications and requirements)	RGp5 (Selection of the right test automation tool)
RSG1 (PLC Test Automation)	✓		✓		✓
RSG2 (Semi-formal requirements for PLCs)				✓	
RSG3 (Evaluation of Proposed Test Automation Approaches)	✓	✓	✓		

test generation and mutation analysis. Moreover, we noticed that improper implementation of test automation for test creation can result in additional costs and effort, and may even be less effective than manual testing in identifying faults, which leads us to formulate R-SG3 for evaluation of the applicability and efficiency of the possible proposed test automation approaches.

Additionally, motivated by filling the research gap RGp4 using R-SG2, we have identified issues in the use of automation, when processing natural language requirements during the development of PLC systems. A mapping of identified research gaps and their connection to formulated research sub-goals of this thesis can be observed in Table 3.2.

3. Propose Solution: The investigation into automating PLC testing in the industry has prompted us to initially tackle the issue of selecting an appropriate test automation tool for practitioners who employ CODESYS IDE for PLC software testing (Paper A). To have a better understanding of the practitioners' needs and preferences in choosing a test automation tool, we have based our research on the Gray Literature Review (GLR) method, introduced by Garousi et al. in [96]. The results of this study have helped us to identify two of the most-discussed test automation tools for CODESYS IDE among practitioners on the web. Subsequently, we have carried out a systematic comparison of these identified test automation tools, utilizing the most crucial test automation features that we extract through a comprehensive literature review. To keep the results of this study in alignment with industrial needs, we have validated these test automation tool features with PLC test engineers of a large automation company in Sweden. As the final step of this study, we have applied both identified test automation tools of CODESYS in a real-world industrial case study.

The existing manual PLC testing process in industry, coupled with the absence of an automated search-based testing tool for PLC programs, has motivated us to introduce an automated PLC to Python translation framework. This framework aims to facilitate PLC programs by enabling automated test case generation. The main goal is to enable automated search-based testing and mutation analysis for PLC programs by leveraging an already existing Python-based testing tool. Rather than developing a new test automation tool for PLC programs from scratch, our approach involves transforming PLC programs into Python scripts using a validated mechanism. To achieve this goal, in this thesis, we propose PyLC, a PLC to Python translation framework that defines the required translation rules, definitions, translation workflow, and translation validation mechanisms required to translate a PLC program in ST/FBD languages into Python based on the IEC 61131-3 standard (Paper B). Next, we facilitate the process of translating a PLC program in FBD language into Python by automating the PyLC fully (Paper D). We evaluated the applicability and efficiency of PyLC by applying it to several different industrial case studies. For automated testing of PLC programs, PyLC leverages Pynguin [38], a well-known Python-based search-based testing tool that supports five different search-based techniques, including MOSA [69], DYNAMOSA [97], MIO [98], RANDOM [99], and WHOLE-SUIT [100] as well as mutation analysis [76].

Further exploration into industrial requirements for effective PLC program testing prompted us to delve into research focused on addressing the issue of using natural language requirement formalization and test creation for PLC programs. To this end, we experiment with transforming three relevant security requirements in industrial libraries into EARS patterns (Paper C). In this experiment, we use three PLC programs corresponding to the selected three requirements. Finally, we develop test cases using the semi-formal requirements of this experiment and evaluate the applicability and efficiency of using this semi-formal notation in PLC requirements engineering and testing.

4. Implement Solution: Driven by the aim to automate PLC testing in an industrial context, we automate the proposed manual PLC to Python transformation called PyLC (Paper D). This involves importing PLC programs written in FBD language, automatically transforming them into Python code while preserving the original program behaviour, validating the accuracy of the transformation, and automating test case generation and execution in both PLC and

Python environments. Moreover, we conduct further evaluations of PyLC's effectiveness by applying it to real-world industrial case studies. These contributions constitute the primary focus of our work. After introducing PyLC as a proof-of-concept solution for transforming PLC programs into Python code to enable automated testing at the unit level, we proceed to develop the initial version of PyLC using Python. This enhanced version of PyLC can parse a PLC program and automatically convert it into equivalent Python code. Furthermore, PyLC incorporates a three-layered unit testing validation mechanism to ensure the accuracy of the translation.

5. Validation: We evaluate the applicability and efficiency of this thesis's contributions by performing testing on real-world industrial case studies after applying the thesis contributions to them. The validity of the gathered results regarding choosing the right test automation tools of CODESYS (Paper A) has been evaluated by conducting a systematic comparison between the identified most-used test automation tools of CODESYS and applying them to a real-world case study and performing unit testing on them. The evaluation of PyLC (Papers B, D) has been done by applying it to several industrial real-world case studies that were completely different in size and complexity. All these case studies are being used on the supervision system of cranes and the volume control system of a large automation company in Sweden, and are all developed in two well-known PLC programming languages of IEC61131-3 standard (i.e., FBD and ST). To validate the accuracy of the PyLC translation, we conduct a series of tests by generating and executing unit-level test cases manually and automatically, using both the Python and PLC versions of the translated PLC programs. The results obtained from the test executions serve as evidence to verify the PyLC PLC in FBD to Python translation framework using testing. This validation is carried out based on the functional requirements of the PLC programs under examination.

The validity of the EARS semi-structured syntax-related contributions in this thesis (Paper C) is evaluated using a controlled experiment and through the generation and execution of unit test cases on three distinct PLC programs. It is important to note that all validation procedures for the contributions of this thesis are carried out at the unit and integration levels, within the CODESYS integrated environment.

6. Research Results: All the results of investigations and evaluations have been published/generated or planned to be published/generated in the form of

research papers and software as shown in Section 4.2.

Chapter 4

Contributions

In this chapter, we present a summary of the contributions towards achieving the overall research goal of this thesis, along with a mapping of each contribution towards the sub-goals.

4.1 Thesis Contributions

The following subsection briefly overviews how the included papers in this thesis contribute towards achieving the formulated research sub-goals.

(I) To meet research sub-goal R-SG1, in paper A [1], we address the practical problem of *choosing the right test automation tool for PLC programs in CODESYS IDE*. In this work, we explore the most popular test automation frameworks of CODESYS IDE by performing a *Grey Literature Review* (GLR) [96] on available test automation frameworks of CODESYS, followed by a qualitative analysis based on several selection criteria. Moreover, we conduct an effective comparison between the identified most discussed test automation frameworks of CODESYS based on 15 important industry-validated test automation features. Finally, we investigate the applicability of the detected test automation frameworks in a real-world case study of an industrial system for crane supervision, by performing its automatic test execution based on two different scenarios. A brief overview of the methodology that is used in this paper can be observed in Figure 4.1.

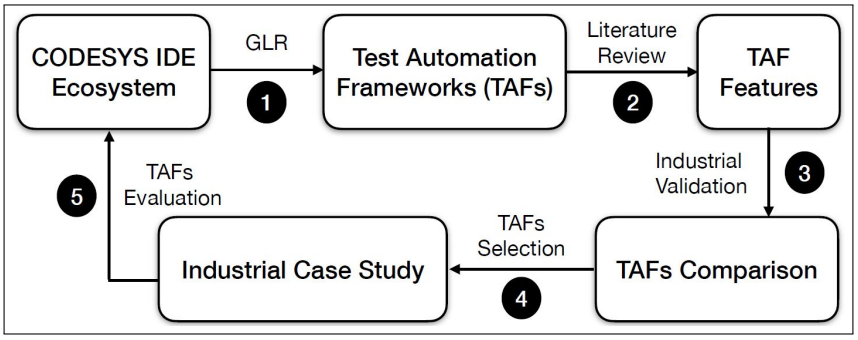

Figure 4.1: An Overview of The Methodology used for Choosing Test Automation Frameworks (TAFs) for PLCs in Paper A.

(II) Next, to achieve research sub-goals R-SG1 and R-SG3, in paper B [2], we propose a *PLC to Python translation framework* called PyLC, which can transform a PLC program written in both Function Block Diagram (FBD) and Structured Text (ST) languages into a Python script, based on different translation rules and unit-testing translation validation mechanisms. A brief overview of the PyLC framework is shown in Figure 4.2. As can be observed in the figure, PyLC proposes the required translation rules based on the IEC 61131-3 standard (step 1 in Figure 4.2), which can be used in the next step to generate the Python code (step 2 in Figure 4.2). Then the generated Python code engages in a three-layered unit-testing translation validation mechanism (step 3 in Figure 4.2) to investigate the correction of the code under translation. Finally, if the translation passes the unit-testing validation mechanism of PyLC, the PLC code is successfully translated into Python (Step 4 in Figure 4.2). The proposed translation rules of the PyLC framework are divided into eight main categories including Input(s), Output(s), Data Type, Data Range, Function Block (FB) behaviour, FB network, Execution Order, and finally, Cyclic Execution. An overview of the translation rules that PyLC adheres to, during the PLC to Python translation process, is shown in Table 4.1.

Moreover, PyLC validates the correctness of the translation based on three validation mechanisms, including:

- Unit testing validation based on requirements.

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

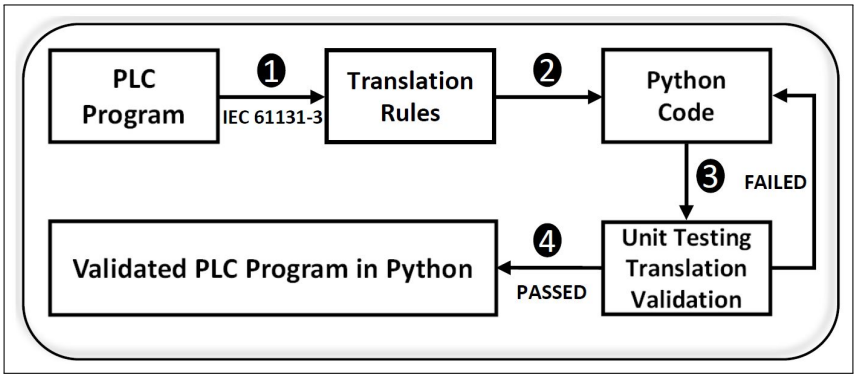

Figure 4.2: An Overview of the PyLC Framework, the Proposed Translation Mechanism for Translating a PLC Program into Python Code and Validating the Translation in Paper B.

- Checking PyLC translation rules.
- Validation using an automatic search-based test generator named Pynguin [38].

We evaluate the applicability and efficiency of our developed translation framework by applying it to 10 different industrial PLC programs. The ultimate goal of conducting this work is to use PyLC to generate search-based test cases for PLC programs during the regression testing phase of the development of industrial control systems.

(III) To achieve the third sub-goal of this thesis, R-SG3, in paper C, we *experiment with requirements engineering and testing* using EARS [31] semi-structured notation for PLC systems. In the requirements engineering part of our experiment, we observe that different individuals prefer different EARS patterns for transforming the same requirement based on their personal interpretations. In the testing part of our experiment, we investigate the applicability of using EARS in the context of PLC testing in two phases. Initially, we execute EARS-based test cases on three PLC programs written in the ST language, developed based on the requirements included in our study. Subsequently, we introduce an EARS-based testing methodology to real-world industrial PLC programs. An overview of the EARS-based requirement specification and PLC

testing methodology used in this experiment is shown in Figure 4.3. As seen, first, We transform the natural language requirements into EARS requirements (step 1 in Figure 4.3), then we concretize the EARS requirements to increase their readability and generate the required test cases for PLC programs under test (step 2 in Figure 4.3). Next, we automatically execute the generated test cases on the PLC programs using the CODESYS Test Manager (step 3 in Figure 4.3). Finally, we check the test results with the expected output to evaluate the efficiency of using EARS in PLC testing (step 4 in Figure 4.3). Moreover, in this work, we propose a semi-automated EARS-based testing methodology for testing real-world industrial PLC programs which starts by extracting the functional requirements of the PLC program via reversed engineering and continues with transforming the extracted requirements into EARS requirements. The procedure follows by transforming the EARS requirements into concertized test cases which are executed automatically in the PLC environment. This process finishes with automated test specification generation via the CODESYS Profiler[1] tool and checking the test execution results. Then we applied the described EARS-based PLC testing methodology to a real-world industrial PLC program and compared it from different testing aspects with its real-world industrial manual testing procedure. The results of this study imply that using the EARS notation in creating requirement-based test cases for PLC programs is promising and can help the PLC testers by establishing an easy-to-understand way of expressing the test specifications.

(IV) To achieve R-SG1 and R-SG3, in paper D, we *automate* the transformation and evaluation of our recently proposed PLC to Python translation framework called PyLC. In this work, we equip the transformation with an incorporated automated XML parser that imports the PLC program in FBD language in the form of a PLCopen XML file. This parser extracts all the necessary information from the file for translation. Additionally, a Python script is employed to automatically write the generated search-based test case values into another PLCopen XML file for test generation. Furthermore, we assess the effectiveness of PyLC by using it in various real-world industrial case studies. The overall automated translation methodology of the PyLC framework is shown in Figure 4.4.

As seen, initially, the PLC program in *PLCopen XML* format is imported to the *Automated XML Analysis* module of the PyLC framework which takes

[1] https://store.codesys.com/en/codesys-profiler.html

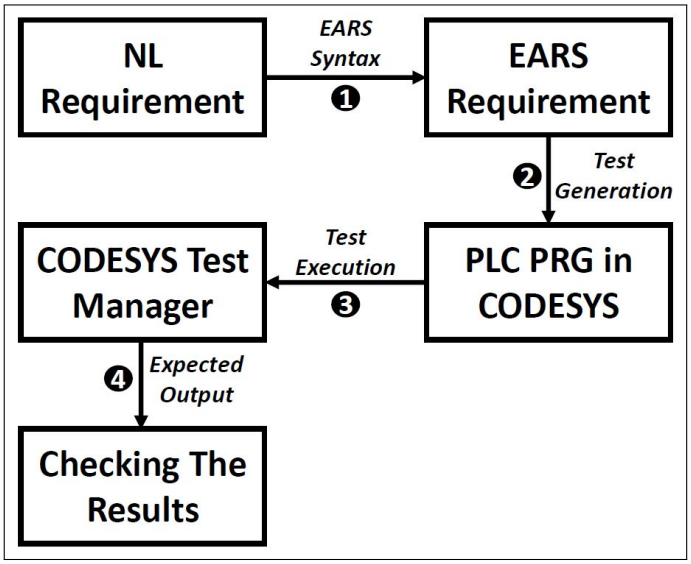

Figure 4.3: An overview of the EARS-based requirement specification and PLC testing methodology.

responsibility for extracting all the required information from the PLC program for translation (Step 1 in Figure 4.4). As the next step, the extracted information from the PLC program is fed into the *Automated Python Code Generation* module which automatically generates an executable Python code based on both gathered information and the behaviour of the PLC program under translation (Step 2 in Figure 4.4). After having the translated PLC program in Python, the next step is to validate the translated PLC program in Python using the *Automated Meta-heuristic Validation* module of the PyLC framework which leverages the *Pynguin* [38] test automation tool (Step 3 in Figure 4.4). The final step is to validate the PLC to Python translation via comparing the results of automated test execution in both Python and PLC environments for the same test cases (Step 4 in Figure 4.4).

To provide a more detailed technical picture of the automated PyLC translation framework, we depicted the detailed methodology of the automated PyLC in Figure 4.5. As seen the first step, is to automatically extract all the required information for translation from the PLC program in the shape of an Open-

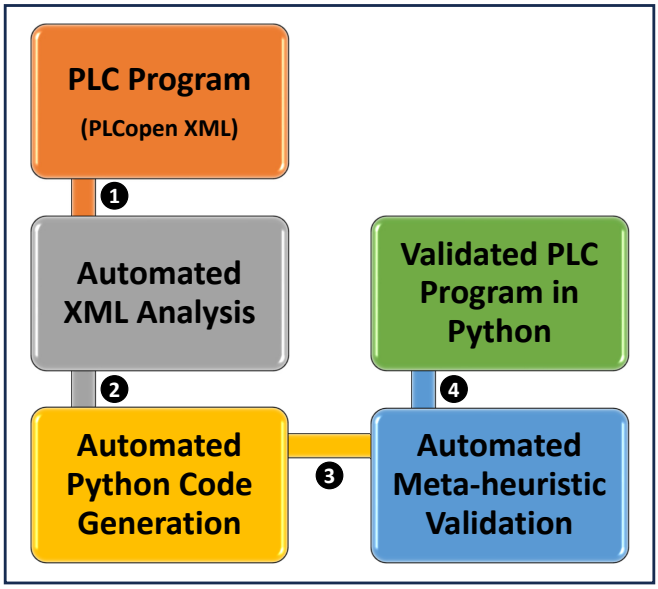

Figure 4.4: An Overview of the Automated PyLC Framework, the Proposed Translation Mechanism for Translating a PLC Program into Python Code and Validating the Translation Automatically.

PLC XML file using the XML analyzer module of PyLC (Step 1 in Figure 4.5). This information is categorised into two main classifications including the Block and POU. The former includes the *Type*, *Position*, *Local ID*, *Network ID*, *Connection*, *Inputs*, *Outputs*, and related *POU* of each existing block in the PLC program under translation whereas, the latter includes several different lists of the *Name*, *Inputs*, *Outputs*, and *Local Variables* of each existing POU in the PLC program under translation.

The next translation step of PyLC is to automatically generate the Python code based on the extracted information in the previous stage using the Python Code Generator module of PyLC (step 2 in Figure 4.5). This includes the automatic generation of Python main and sub-functions, *Inputs*, *Outputs*, *the FBD network*, and doing the *Data Type Conversions* from PLC to Python. The next step of translation is the first stage of an attempt to validate the translation of the translated code in Python using meta-heuristic algorithms and mutation

analysis. To this end, the generated code in Python needs to be imported into the meta-heuristic test generation module of PyLC which is assisted by the Pynguin tool [38] (step 3 in Figure 4.5). After gathering the results of the automated search-based testing of the translated PLC program in Python using the Pynguin tool, the test results are recorded and the same test cases are transferred into the CODESYS IDE via CODESYS Test Manager tool to be executed on the original PLC program in PLC environment (step 4 in Figure 4.5). The final step of the PyLC translation mechanism is to compare the results of executing the search-based generated test cases in both PLC and Python versions of the PLC program to observe whether they produce the same results or not (step 5 in Figure 4.5). If the execution of the same test cases in both PLC and Python variations of the PLC program generates the same results, the translation of the PLC program to Python is considered valid, otherwise it's not. In the future, We aim to delve deeper into evaluating the efficiency and effectiveness of different search-based algorithms for PLC testing. More technical details about the automated PyLC translation framework are illustrated in Section 11.

4.1.1 Individual Contribution

I am the primary researcher, driver and author of all the included papers. However, all the other co-authors have contributed with their valuable ideas, discussions and reviews. The supervision team has also contributed to refining the text.

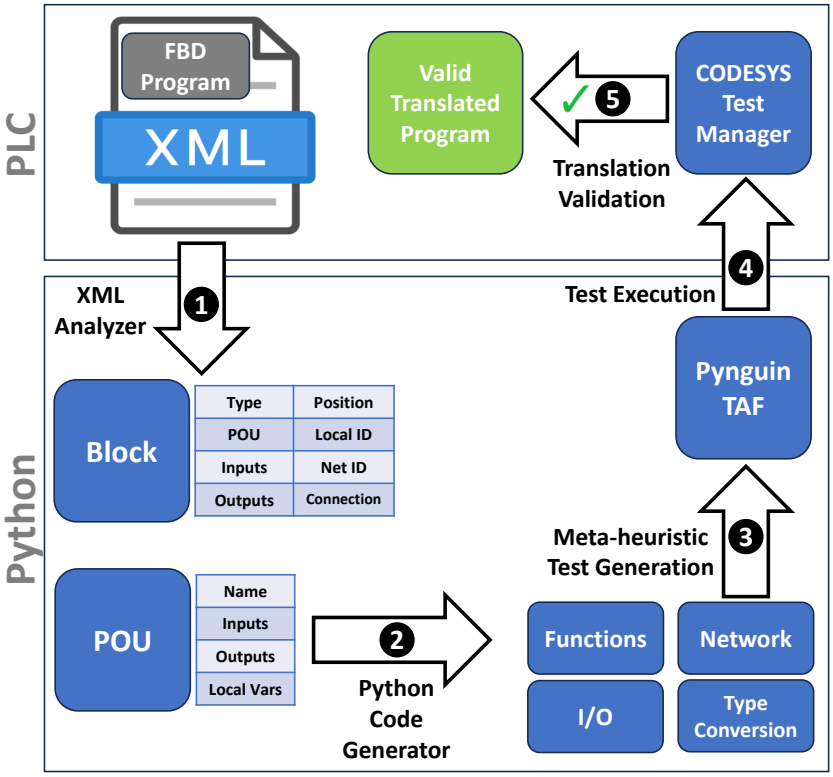

Figure 4.5: A Detailed Overview of the Automated PyLC Framework for Translating a PLC Program into Python Code and Validating the Translation Automatically.

4.2 Included Papers

All the included papers in this thesis have a contribution towards meeting the overall research goal, and the mapping of these contributions to research sub-goals is shown in Table 4.2.

(R-SG1) is achieved using three included papers, that is, Papers A, B, and D. In particular, Paper A contributes towards realizing the R-SG1 by identifying the most popular test automation tools of one of the well-known PLC IDEs, called CODESYS. This work addresses the non-trivial problem of choosing the

right test automation tool for PLC programs, by conducting a systematic Grey Literature Review (GLR)[2] followed by a comparison between the identified popular test automation tools of CODESYS based on the industry-validated features. Next, Paper B addresses R-SG2 by proposing a PLC to Python translation framework called PyLC, with the ultimate goal of bringing the benefits of a Python-based automated search-based test generator to PLC testing. PyLC proposes the required translation rules, workflows, and unit-testing translation validation mechanisms for translating a PLC program into an executable equivalent Python code. The final contributions towards achieving R-SG1 are provided by Paper D, which is an effort to fully automate PyLC, by enabling it to automatically import a PLC program described in FBD language, and transform it into the equivalent executable Python code followed by validating the translation correction via an automated meta-heuristic testing approach and mutation analysis. Moreover, Paper D addresses R-SG1 by evaluating the applicability and efficiency of PyLC using several real-world industrial PLC case studies.

Achieving (R-SG2) of this thesis has been realized using the contributions proposed in Paper C. This paper investigates the applicability and efficiency of using a well-known semi-formal requirement notation called EARS [31] in the context of PLC engineering and testing. This paper consists of two experiments that investigate both engineers' and testers' ways of using EARS notation for transforming the requirements and testing the PLC programs. This paper also proposes a semi-automated EARS-based PLC testing mechanism for real-world industrial PLC programs and investigates the efficiency and applicability of this mechanism by applying it to a real-world PLC program and comparing it with the current manual testing procedure in industry.

(R-SG3) which focuses on the evaluation of the proposed test automation approaches, is met using the provided contributions of all included papers of this thesis (Papers A to D). Paper A addresses this sub-goal by applying both identified test automation tools of CODESYS on two real-world PLC programs. Paper B and Paper D contribute towards this sub-goal by applying the proposed test automation tool, PyLC, to 20 different real-world case studies in the context of industrial control systems of the port cranes and nuclear plants. Finally, Paper C helps to achieve R-SG3 by using the EARS notation for testing

[2]GLR is a method for reviewing the literature while excluding the academic results to identify the practitioners' point of view regarding a specific subject.

four different PLC programs and comparing it with the current existing manual testing procedures in the industry from different testing perspectives.

A summarized mapping of included research papers concerning their contribution to each R-SG, respectively, is shown in Table 4.2.

Table 4.2: Mapping of research papers w.r.t. their respective contribution to Research Sub-goals (R-SG)

	R-SG1 (PLC Test Automation)	R-SG2 (Semi-formal requirements for PLCs)	R-SG3 (Evaluation of Proposed Test Automation Approaches)
Paper A	✓		✓
Paper B	✓		✓
Paper C		✓	✓
Paper D	✓		✓

4.2.1 Paper A

Title: Choosing a Test Automation Framework for Programmable Logic Controllers in CODESYS Development Environment.

Authors: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu.

Status: Proceedings of the 15th IEEE International Conference on IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2022), The Next Level of Test Automation (NEXTA 2022), 2022. Publisher: IEEE.

Abstract: Programmable Logic Controllers are computer devices often used in industrial control systems as primary components that provide operational control and monitoring. The software running on these controllers is usually programmed in an Integrated Development Environment using a graphical or textual language defined in the IEC 61131-3 standard. Although traditionally, engineers have tested programmable logic controllers' software manually, test automation is being adopted during development in various compliant development environments. However, recent studies indicate that choosing a suitable test automation framework is not trivial and hinders industrial applicability. In this paper, we tackle the problem of choosing a test automation framework for testing programmable logic controllers, by focusing on the COntroller Development System (CODESYS) development environment. CODESYS is deemed popular for device-independent programming according to IEC 61131-3. We explore the CODESYS-supported test automation frameworks through a

grey literature review and identify the essential criteria for choosing such a test automation framework. We validate these criteria with an industry practitioner and compare the resulting test automation frameworks in an industrial case study. Next, we summarize the steps for selecting a test automation framework and the identification of 29 different criteria for test automation framework evaluation. This study shows that CODESYS Test Manager and CoUnit are mentioned the most in the grey literature review results. The industrial case study aims to increase the know-how in automated testing of programmable logic controllers and help other researchers and practitioners identify the right framework for test automation in an industrial context.

4.2.2 Paper B

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

Authors: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu.

Status: Proceedings of the 38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023), 2023. Publisher: ACM.

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.

4.2.3 Paper C

Title: An Empirical Investigation of Requirements Engineering and Testing Utilizing EARS Notation in PLC Programs.

Authors: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu.

Status: Submitted to the Springer Nature Journal's Special Issue on Topical Issue on Advances in Combinatorial and Model-based Testing 2023, under review.

Abstract: Regulatory standards for engineering safety-critical systems often demand both traceable requirements and specification-based testing, during development. Requirements are often written in natural language, yet for specification purposes, this may be supplemented by formal or semi-formal descriptions, to increase clarity. However, the choice of notation of the latter is often constrained by the training, skills, and preferences of the designers.

The Easy Approach to Requirements Syntax (EARS) addresses the inherent imprecision of natural language requirements with respect to potential ambiguity and lack of accuracy. This paper investigates requirements specification using EARS, and specification-based testing of embedded software written in the IEC 61131-3 language, a programming standard used for developing Programmable Logic Controllers (PLC). Further, we study, by means of an experiment, how human participants translate natural language requirements into EARS and how they use the latter to test PLC software. We report our observations during the experiments, including the type of EARS patterns participants use to structure natural language requirements and challenges during the specification phase, as well as present the results of testing based on EARS-formalized requirements in real-world industrial settings.

4.2.4 Paper D

Title: PyLC 2.0: An Automated Framework for Transforming and Validating PLC Software using Python and Pynguin Test Generator.

Authors: Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu.

Status: Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC 2023). Publisher: IEEE.

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.

Chapter 5

Results

This section briefly overviews the gathered results of the included papers in this thesis including Papers A to D. We start by summarizing the results of choosing the right test automation tool for one of the most common IDEs for PLCs in the industry, CODESYS. Then we discuss the results of applying the PyLC translation framework to ten different industrial PLC programs followed by representing the results of using EARS notation for PLC testing. Finally, we summarise the gathered results of the automated translation of FBD programs to Python using the fully automated variation of the PyLC translation framework applied to 10 different real-world industrial PLC programs. More detailed results of each included papers in this thesis can be observed in Sections 8, 9, 10, and 11.

5.1 Choosing the Right Test Automation Tool for CODESYS IDE

Paper A [1] of this thesis is an effort to assist practitioners in choosing the right test automation tools for CODESYS PLC IDE through a hybrid methodology as described in Figure 4.1.

5.1.1 Discovered Test Automation Frameworks of CODESYS IDE

As a result of the GLR, we obtained 120000 search results, all written in English. We only stored the first 100 results locally to build the pool of contents since we discovered that these contain relevant sources to our topic. Most of the objects in the final version in the pool of objects have been published by industry individuals, including IDE developers and PLC vendors. Aiming to establish a trade-off between the preferences of companies and independent framework developers in our results, we included valid third-party developers and GitHub topics in the final pool. After reviewing the content of the pool, we ended up with a pool consisting of 13 sources. After analyzing the final objects based on the defined criteria, we discovered three test automation frameworks as the most prevalent automation frameworks targeting CODESYS. Out of all the collected results, 62% of the objects in the pool are pointing towards CODESYS Test Manager[1] (the largest share of the discovered objects). Two other frameworks, CoUnit[2] and TcUnit are revealed in 15% of the objects each. Other frameworks were mentioned in 8% of the objects. Our results suggest that most of the discovered objects of our GLR after screening and applying the selection criteria point towards CODESYS Test Manager, CoUnit (formerly known as CfUnit), and TcUnit as the predominant test automation frameworks targeting CODESYS. We note here that CoUnit is developed based on TcUnit and both frameworks have similar functionality. Since CODESYS IDE officially supports only the former, we include CoUnit in the final list of discovered automation frameworks.

Our results suggest that the most prevalent test automation frameworks targeting CODESYS IDE for PLC testing are the CODESYS Test Manager and CoUnit.

5.1.2 Test Automation Frameworks Features

First, we need to identify the essential features of these test automation frameworks before conducting a comparison between the discovered test automation frameworks of CODESYS. To this end, we followed a hybrid approach

[1] https://store.codesys.com/codesys-test-manager.html?
[2] https://forge.codesys.com/lib/counit/home/Home/

Table 5.1: Extracted and Validated Framework Features.

Industry-validated Features		
Category	**Feature**	**Extraction Source**
Company Constraints	Cost	[101], [102], [103], [104], [105]
	Supported Platforms	[101], [102],[103]
Maturity	Industrial Usage	[101]
	Stage of Development	[101]
Testing Functionalities	Documentation and Report Generation	[101]
	Playback Record	[103]
	Test Suite Support	[105]
	Test Suite Extension	Industry
Tool Flexibility	Teamwork Support	[101]
Usability	DevOps/ALM Integration Support	[102]
	Continuous Integration (CI) Support	[102]
	Script Language	[102], [103], [105]
	Availability of Customer Support	[101], [102]
	Quality of Documentation	[101]
	Maintenance Support	Industry

which consisted of a literature review of related works followed by an industrial feature validation. Based on three sources of information used (academic works, industrial input, and official documentation), we discovered 29 industry-reported essential features that should be considered when choosing a test automation framework for PLCs. We acknowledge that many of these features are generic. Still, the instantiation of these features is specific to PLCs. We divided the discovered features into five categories based on their focus, including Company Constraints, Maturity, Testing Functionalities, Framework Flexibility, and Usability. Since our aim in conducting this work is to address the needs of industrial practitioners, we evaluated the validity of the discovered features by checking them with a group of engineers working with CODESYS and PLC testing in an industrial automation company in Sweden. These engineers validated these features of a test automation framework by marking the ones a tester would use to choose such a framework (i.e., 15 out of 29 features were considered important by these engineers). The list of the discovered and validated test automation framework features and non-validated ones as well as their category and source of extraction, are shown in Table 5.1 and 5.2 respectively. It should be noted that the gathered data does not need any further processing (e.g., open coding).

Table 5.2: Other Extracted Framework Features.

Other Features		
Category	**Feature**	**Extraction Source**
Company Constraints	Ease of Installation	[101], [102]
	License Type	Tool Documentation
Testing Functionalities	Test Script Specification	[101]
	Supported Testable Objects	Tool Documentation
	Requirements Traceability	[101]
	Script Creation Time	[102]
	Import Support	[105]
Tool Flexibility	Backward Compatibility	[101], [105]
	Standard Input Format	[101]
	Modularity of The Tool	[101]
	Framework Development Language	[105]
Usability	Programming skills	[102], [103]
	Report Format	[103]
	Graphical User Interface (GUI)	[101]

We have discovered several features that should be considered when choosing a test automation framework for PLC testing: cost, supported platforms, industrial use, stage of development, documentation and report generation, record playback, test suite support, test suite extension, team support, DevOps/ALM support, continuous integration support, scripting language, import support, availability of customer support, quality of documentation, and maintenance support.

5.1.3 Test Automation Frameworks

We conducted an initial comparative examination given the features identified in the previous section. We focus on Test Manager and CoUnit as our chosen test automation frameworks in CODESYS IDE. The results of this comparative examination are shown in Table 5.3. Even if both frameworks support only Windows platforms and have continuous integration support, we can observe significant differences. In terms of cost, CODESYS Test Manager is a commercial product but available for academics to use in their research. CoUnit is an open-source software freely available. Industrial usage of Test Manager

is considered HIGH since its use has been reported in several industry-related reports [106], [107], [108], [109], [110].

Regarding the framework's maturity, CODESYS Test Manager seems to be more mature and has evolved through eight different versions so far, compared to CoUnit (i.e., in 3 versions). One of the main advantages of CODESYS Test Manager is the ability to record and playback that is not supported by the counterpart framework. Both frameworks support test suites in .xml file format. In addition, CODESYS Test Manager has the advantage of supporting .tsd (Tamino Schema) extension (used as a container of elements that a Tamino XML Server document contains).

CODESYS Test Manager supports test suites to be extended by using specific predefined test commands, but the extension of test suites in CoUnit demands ST programming knowledge, and the user needs to instantiate the code for each single test case. CODESYS Test Manager supports Python and all IEC 61131-3 programming languages for developing the test scripts, while CoUnit only supports the ST programming language. Availability of customer support is another essential factor from an industry point of view in this comparison, and the CODESYS Test Manager seems to be superior in this respect. The quality of the documentation provided by the CODESYS Test Manager is excellent since comprehensive educational material and good video tutorials are available. On the other hand, CoUnit provides less documentation and tutorials. Maintenance support is another important feature proposed. CODESYS Test Manager supports direct main PLC program testing and one instantiation of the code under test can be used in all related test suites but these features are not available in CoUnit.

Based on our initial comparison between CODESYS Test Manager and CoUnit based on the 15 industry-validated features, the results show that CODESYS Test Manager is more mature and has several advantages over CoUnit, including user support, record and playback features, and easy test suite extension. Nevertheless, CoUnit, as an open-source counterpart, also provides testers with many key features used during PLC testing.

Table 5.3: An Overview of the Comparison between CODESYS Test Manager and CoUnit based on the Validated Features.

Feature	Test Manager	CoUnit
Cost	MIX *Commercial license, but free to use for academic purposes	FREE *Open Source license
Supported Platforms	Microsoft Windows	Microsoft Windows
Industrial Use	HIGH	LOW
Stage of Development	MATURE *8 versions released so far	PARTIAL *3 versions released so far
Documentation and Report Generation	COMPLETE	SUMMARIZED
Playback Record	YES *Can be realized via the Test Progress feature	NO
Test Suite Support	YES *.tsd, .xml extensions are supported	YES *.xml extension is supported
Test Suite Extension	EASY *New test cases can easily be developed using the available graphical test commands. *One instantiation of the POU under test can be used in all new test suites and test cases *The number of test cases inside a test suite is not limited	HARD *New test cases need to be developed in ST language *For every new test case a new distinct instantiation of the POU under test is required *Every test suite can only contain 100 test cases
Teamwork Support	NO	NO
DevOps/ALM Integration Support	No Information Provided	No Information Provided
Continuous Integration (CI) Support	Yes	Yes
Script Language	Python, All IEC 61131-3 Supported Programming Languages	Structured Text (ST)
Availability of Customer Support:	YES(Official CODESYS customer support is available)	NO
Quality of Documentation	VERY GOOD *Both tool documentation and official tutorial videos are available online	GOOD *Tool documentation and textual tutorial are available online
Maintenance Support	EASY *Direct testing of the PLC main program is supported *GUI and graphical test commands are available	HARD *Direct testing of the main PLC program is not supported *GUI and graphical test commands are not available

5.1.4 Applicability in an Industrial Case Study

Aiming to answer this research question, we applied the two test automation frameworks we found through our GLR to an industrial case study by considering several possible test scenarios. Our case is a control system provided by a large automation company in Sweden consisting of several POUs. This system is developed in the FBD programming language.

The Function Block (FB) in this POU consists of several computational blocks executed cyclically. The program executes in a cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs). The FBD program is created as a composition of interconnected blocks with data flow communication. When activated, a program consumes one set of input data and then executes it to completion. We considered functional scenarios for testing the POU.

We evaluate this functionality and the applicability of *Test Manager* and *CoUnit* by automating the test execution for the provided case and all POUs. To this end, we generated several test suites consisting of manually created test cases.

We report our overall experiences in using both test automation frameworks. The following results and features are PLC-specific. Regarding **installation and configuration**, we found out that setting up CODESYS Test Manager seems to be more straightforward since it can be installed as a standard add-on package. On the other hand, CoUnit needs to be installed as a package and imported as a library in every project under test. Regarding the **ease of use**, CODESYS Test Manager is more user-friendly and provides features for developing test scenarios using available test commands in the GUI integrated into CODESYS IDE. Moreover, developing test cases with this framework does not require the use of any of the IEC61131-3 programming languages. On the other hand, creating the same test cases in CoUnit is more time-consuming due to the use of ST scripts and instantiations. When comparing the frameworks' capabilities related to **testable objects**, we found out that the Test Manager can create harnesses for PLC applications, IEC libraries, and communications. In contrast, CoUnit can only be used at the application level. Regarding **test assertion timeouts** we note here that PLC programs are executed cyclically in a loop, and one needs to set a test assertion timeout to make sure that the result

comparison process ends after a certain amount of time. Only CODESYS Test Manager can be used to set a custom timeout, a useful feature when testing complex PLC programs. After executing test scripts on both frameworks, we discovered that test reports generated by CODESYS Test Manager provide the user with detailed information. On the other hand, CoUnit only reports scarce information.

Using the discovered features as a basis, the application on an industrial PLC program revealed that both frameworks provide proper automation functionality. However, CODESYS Test Manager seems to be more mature, provides more helpful test execution features, and is more user-friendly. In contrast, CoUnit seems limited in its usefulness, and working with it requires ST programming.

5.2 Translation of ST/FBD Programs to Python

Paper B [2], as the second included paper in this thesis is our first step towards enabling the Python-based automated search-based testing for PLC programs by translating them into executable Python code. This translation framework is called PyLC. PyLC follows the mechanism depicted in Figure 4.2 and adheres to the translation rules in Table 4.1.

5.2.1 PyLC Translation

We consider ten different PLC programs to evaluate our proposed translation framework in real-world circumstances, including six ST and four FBD programs. Detailed information on the translated PLC programs is shown in Table 5.4. 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.

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

We note here that, according to the data in Table 5.4, 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.

5.2.2 PyLC Validation

To evaluate the proposed method, we use the translation results of the translated PLC programs in Table 5.4 by three different unit testing mechanisms described in Figure 9.6. In the following subsections, we describe and demonstrate the results regarding each unit testing validation step, respectively.

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.

Based on the proposed technique for this type of validation (as shown in

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

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

Figure 9.6), we analyze the behaviour of the translated code from two different aspects, which 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*[3]. Table 5.6 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 5.5 shows more details and results of testing these blocks. As it can be observed in Table 9.4, 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 CODESYS IDE, indicating the program's accurate translation using PyLC

[3]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 5.7: An overview of the results of Test Case (TC) execution on 10 cases based on the proposed PyLC Translation Rules

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.

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 5.7.

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 [97] (Pynguin's default algorithm) as our algorithm of choice for generating test cases due to its dynamic nature, multi-objective optimization capabilities, efficient search space exploration, adaptability, scalability, and efficient resource utilization.

We have followed Pynguin's default configuration using DYNAMOSA, a test generation time budget of 10 minutes, 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 5.8.

As seen in Table 5.8, 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[4]. We observe that Pynguin starts mutation analysis for 9 out of 10 PLC programs, and in all except one case, it can kill all the mutants. The results seem to be influenced by the 10-minute 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 vali-

[4]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 5.8: Results of Automatic Test Generation/Execution for Translated PLC Programs using Pynguin TAF

dating 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.

5.3 Application of EARS Notation in Testing PLCs

The applicability and efficiency of using a popular and scientifically proven notation such as EARS in the context of PLC testing are investigated by us in Paper C [3]. The overall methodology of this work is depicted in Figure 4.3. This work consists of two main parts including requirement engineering experiment and PLC testing. We briefly overview the gathered results for both sections in the following.

Table 5.9: The natural language requirements used during the experiment.

Requirement ID	Requirement Text
RI1	User account should be uniquely identified to a user.
RI2	The software shall warn the user of malware detection.
RI3	Only authorised devices are allowed to connect into the ICS network

5.3.1 Requirement Engineering Results

We investigated the industrial libraries provided by a large-scale company focusing on the development and manufacturing of control systems. We identified three candidate requirements matching our criteria, shown in Table 5.9. The requirements should not be trivial, yet fully manageable to use within 60 minutes and no domain-specific knowledge should be needed to understand the requirements. We then assessed the relative difficulty of the identified requirements by manually writing and creating tests. For each requirement, we have collected data about the type of EARS template used by each participant, the approaches, and the challenges participants experienced during requirement representation using the EARS notation. The results are shown in Table 5.10, Table 5.11, and Table 5.12.

Participants strictly adhered to one or multiple EARS templates. It seems that the ubiquitous template has been used by all participants to model requirement RI1 and just in one case when representing requirements RI2 and RI3 (as shown in Table 5.10). Participants explained that the ”shall” statement is clearly indicated and should be used to describe the required behaviour. Nevertheless, one participant decided to use the unwanted behaviour template for RI1 to indicate the prohibited behaviour in such a form that can be used for testing.

The event-driven and unwanted behaviour templates have been used by participants to represent requirement RI2, while some participants used the state-driven pattern (as shown in Table 5.11). Participants chose to do this since they drafted requirements in several increments. Firstly, they considered how the system behaves typically (also called sunny-day behaviour). For some

Table 5.10: Results of the templates used for each requirement used in the experiment.

RI1	RI2	RI3	Requirement ID/EARS Template
10	1	1	**Ubiquitous (U)**
0	5	4	**Event-Driven (ED)**
1	5	6	**Unwanted Behaviours (UB)**
0	0	3	**State-Driven (SD)**
0	0	0	**Optional Features (OF)**

participants using EARS, this results in requirements in the state-driven and event-driven patterns. Secondly, some participants decided to specify what the system must do in response to the unwanted behaviour, which produced requirements in the unwanted behaviour pattern.

In addition, the thematic analysis of the notes taken by participants when performing these steps in requirement representation resulted in several main themes related to approaches and challenges experienced during the translation process. Several participants mentioned that the initial NL requirements are not complete and clear such that these can be used directly for testing. One participant mentioned the following: *"What happens if the device is not authorized, missing failure models, startup/default/safe state...?"*. This resulted in issues when starting with the translation process, especially when deciding which templates to use. Several participants had issues in deciding when to use single or multiple EARS templates to cover both positive and negative behaviours that need to be tested. One participant stated the following: *"We could possibly use event-driven type requirement. At the same time, it is unwanted we could use, this one is quite complicated"*. Some participants preferred the use of the "shall not" form, which has been observed by some participants as having an impact on the test case created since only a set of test cases involving the unwanted behaviour would need to be created to show satisfaction with the requirement. Another observation relates to the use of an optional feature template, which for the given requirements was not used by any of the participants since there was no need to specify any product variation or specific features.

Table 5.11: Results of the requirements writing in terms of the templates used by each participant for each requirement. EARS template types are shown using their specific acronyms as stated in Table 5.10.

RI1	RI2	RI3	Requirement ID/Participants
U, UB	U, UB, ED	U, SD, ED	**P1**
U	ED	UB	**P2**
U	ED	UB	**P3**
U	UB	SD	**P4**
U	ED	UB	**P5**
U	ED	UB	**P6**
U	SD	UB	**P7**
U	UB	ED, UB, SD	**P8**
U	UB	ED	**P9**
U	UB	ED	**P10**

Test Results of *PRG1*

PRG1 is the PLC program we considered for testing the RI1 requirement (refer to Table 5.9) in the PLC environment. This program is using the values of the *user account* and *user* lists. Then it checks for unique IDs and returns an indication of whether each user account is uniquely identified to a user or not. A snippet of the PRG1 PLC program is shown in Figure 10.2.

To design and execute the required test cases to test the RI1 Requirement in PRG1, we use the transformed requirement from the NL requirement shown in Table 5.13.

Based on the EARS requirement we use two test cases to cover the identification of the user and the case when the user is not identified. Each test case includes the following three test actions: two *WriteVariable* test actions to alter the *user* and *user account* inputs and one *CompareVariable* test action that compares the actual output with the expected one. The generated test cases for PRG1 used to test the adherence of the program to RI1 requirements are shown in Figure 10.3.

After designing the required test cases, we execute them automatically on PRG1 to investigate the adherence of the mentioned PLC program to the RI1 requirement. As can be observed in Figure 10.4, all test cases have been exe-

Table 5.12: Results showing the main themes identified related to approaches and challenges encountered during the translation process.

Main Themes	**Theme Descriptions**
Requirements are not complete and clear enough for EARS translation.	When starting with the translation, requirements in NL are not complete enough to decide precisely which EARS template to use.
Using single or multiple EARS templates is not clear enough, especially when using these for testing.	There is a need, when using these patterns for testing, to use multiple and separate templates for each requirement to cover both positive and negative cases arising.
The system perspective is not easily identifiable from the requirements.	It is difficult to decide which perspective to use when translating the EARS requirement (e.g., system, subsystem level).
The optional feature template is not applicable for the selected requirements	Even if the Option requirement is used for systems that include a particular element and variants, this modelling form was not used during requirement transformation using the EARS notation since the participants did not need to handle system or product variation.

Table 5.13: EARS Requirements examples obtained from the experiment and the resulting concretized EARS requirements.

Requirements	EARS Requirements	Concretized EARS Requirements
RI1	The <user account system> shall <identify the user> If <the user is not identified> then <user account system> shall <alert>	if <uniqueID=FALSE> then <UniqueUserAccount> shall <Result_Unique=FALSE>
RI2	When <malware is detected> the <system> shall <warn the user>	When <NormalActivity ≠ MaliciousActivity> the <MalwareDetection> shall <MalwareDetected=TRUE>
RI3	When <the device is authorised> the <system> shall <grant access to the device>	When <found=TRUE> the <SearchID> shall <ConnectionAllowed=TRUE>

cuted in 0.3 seconds. All executed test cases have successfully passed on the PRG1 program.

Test Results of *PRG2*

The PLC program we use for executing the generated test cases for *RI2* in Table 5.9 is named PRG2. This program is shown as a black-box malware detection system in the PLC environment that can be used for investigating the context of RI2. PRG2 consists of the following interfaces: two input signals named *MaliciousActivity* and *NormalActivity* as well as one output signal named *MalwareDetected*. When *MaliciousActivity* and *NormalActivity* signals have divergent information, the Malware Detection system is triggered, and the value of the *MalwareDetected* signal becomes True. An interface snippet of PRG2 is shown in Figure 10.5.

Considering the results of the experiment we use the resulting EARS *Event-driven requirement* pattern as the most suited type of template for transforming the requirement from NL to EARS in the form shown in Table 5.13.

Based on the developed EARS requirement for RI2 requirement, we generate two test cases for PRG2. Each test case consists of two test actions (*Ma-*

liciousActivity and *NormalActivity*) that alter the value of the inputs, as well as one test action (*Expected Output*) that compares the actual behaviour with the expected one. The first test case checks if a (*Malware is Detected*) while the second test case checks if a (*Malware is Not Detected*). The generated test cases for PRG2 based on the RI2 requirement are then automatically executed using CODESYS Test Manager in 1.71 seconds. All developed test cases have successfully passed.

Test Results of *PRG3*

PRG3 is the PLC program used to execute the generated test cases for *RI3* in Table 5.9 (*"Only authorised devices are allowed to connect into the ICS network"*. This program consists of the following units: *1)* a database of authorised device IDs, which is implemented using an array of IDs, *2)* an input signal corresponding to the device ID that needs to be authorised, and *3)* a boolean output signal (i.e., *found*) which returns True in the case of the authorised device being allowed to connect given the ID is known. We show a snapshot of this PLC program in Figure 10.6.

As discussed in Section 5.3.1, different individuals transformed the NL requirement into the EARS requirement in different forms. We use the most common form developed by the participants to transform RI3 to an EARS *Event-Driven* syntax pattern in the following form shown in Table 5.13.

Based on the aforementioned EARS requirement for RI3, we developed 2 test cases for *Successful Authorization* and *Unsuccessful Authorization*. Each developed test case consists of two actions, including the provision of a *new Input ID* and *Comparing the actual output with the expected output*. The generated test cases have been automatically executed on PRG3 using CODESYS Test Manager in 1.14 seconds. Both test cases have successfully passed after being executed on the PRG3 PLC program.

Aiming at evaluating the applicability of using EARS semi-structured syntax when creating test cases for PLC programs, we used three programs that implement the behaviour stated in the three provided natural language requirements used in this experiment. All these three PLC programs are developed in CODESYS IDE using the ST programming language. In this work, we refer to these programs as *PRG1*, *PRG2*, and *PRG3*. After generating the EARS-based test cases for each program, we execute these automatically using the

CODESYS test automation framework named CODESYS Test Manager[5]. The final step in this methodology is to compare the actual output with the expected output to observe whether the program works as expected.

We used the concretization steps of the EARS expressions as stated by Flemstrom et al. [111]. This happens by mapping the system response, condition, and events to the actual implementation in PLC. This contains information about the implementation elements of a system and its interfaces. An engineer needs to consider this information and identify the given signals and their characteristics. In this way, we define a set of signals related to the feature under test. In these cases, the next step for the selected requirements would be to design test cases to show that the requirement has been met. In our experiment, we could directly use a subset of positive and negative cases by randomly choosing values from an equivalence class. Nevertheless, in a general case, the translation and concretization steps are not easy and one would need to decide how to automate such steps and if we are to use exhaustive testing, equivalence class testing, combinatorial testing, or any other test selection technique for designing test cases.

5.3.2 EARS-based Testing vs Manual PLC Testing

Comparing the overall current industrial manual testing process of a real-world PLC program (shown in Figure 10.8) versus the proposed semi-automated industrial EARS-based testing mechanism of this thesis as depicted in Figure 10.7, reveals several facts including:

1. **Need of domain-specific knowledge.** One needs to have a good understanding of one of the IEC61131-3 programming languages to be able to develop test cases for the PLC program in the current industrial approach. Moreover, the manual tester needs a testing background and engineering experience to implement and connect all testing units properly. On the other hand, testing the PLC programs with the proposed mechanism using CODESYS Test Manager does not demand any deep knowledge of specific programming language and can be handled easily using *Test Actions*.

[5]https://store.codesys.com/en/codesys-test-manager.html

2. **Efficiency.** In the case of a simple PLC program such as *CraneNumberCheck* which consists of 25 Lines of Code (LOC), the test script consists of 119 LOC which shows a difference in efficiency. On the other hand, the proposed EARS-based testing approach only consists of 26 test actions, which use all the powerful features of CODESYS IDE.

3. **Manual overhead and complexity.** The current testing process for PLC programs in the industry is highly complex, with significant manual intervention. Specifically, many features already present in the CODESYS IDE, such as cyclic execution, delay, test control process, and test start trigger, are being recreated manually. This redundancy exacerbates the complexity, especially with more intricate PLC programs. Conversely, the proposed EARS-based testing approach simplifies this by requiring a manual definition of only the inputs and expected outputs. All other features are readily accessible through the user-friendly GUI of the CODESYS Test Manager tool. Additionally, the availability of predefined test actions within the Test Manager tool enhances the use of CODESYS's features and automation capabilities for PLC testers.

4. **Test specifications.**The existing manual testing process in the industry offers testers limited information, providing only the outcomes of passed or failed test cases. In contrast, the proposed EARS-based testing approach utilizes both CODESYS Test Manager and CODESYS Profiler to provide a comprehensive set of test specifications. These include additional details like test execution time, coverage reports, outcomes of individual test actions, test verdicts, and more.

5. **Ambiguity and clarity of functional requirements.** After reviewing a limited set of requirements gathered from the industry, it became apparent that the current functional requirements are predominantly at the system level, lacking specificity for individual code branches. Additionally, the complexity of industrial testing processes relies heavily on the tester's expertise. In contrast, the proposed EARS-based approach reduces the vagueness of requirements and encompasses both unit and system-level testing, potentially leading to a more thorough testing procedure. Furthermore, this approach yields requirements and test cases that are straightforward and comprehensible, facilitating understanding among all stakeholders, including testers, managers, and clients.

5.4 Automated Translation of FBD Programs to Python

In the final step of this thesis towards enabling and facilitating automated PLC testing for PLC programs, we fully automated the PyLC translation framework using the depicted methodology in Figure 4.5 in Paper D [4]. The results of applying automated PyLC to 10 different real-world industrial FBD programs are briefly explained in the following.

5.4.1 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 5.14. 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 5.14, 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 5.14, we can draw several conclusions. First, the FBD programs selected for translation encompass a variety of FBD block types, as detailed in Section 5.14. 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.

> *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 within industrial automation but also offers a systematic approach

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 5.14: Information Regarding the Translated PLC Programs (PRG) in FBD language into Python using PyLC

to bridge the gap between PLC programming languages and general-purpose languages like Python.

5.4.2 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 [38] 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 5.15. 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 5.15, 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 behaviour. Iteration counts vary 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

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 5.15: Information Regarding Automated Testing of The Translated Real-world PLC Programs to Python using the Pynguin Tool

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.

The automated 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.

Chapter 6

Discussion and Limitations

In this chapter, we present and justify the chosen techniques and coverage criteria, PLC to Python code translation, selection of Python as translation destination language, and limitations of our study such as focusing on a specific IDE (CODESYS) and testing the *Timer* blocks which are planned to be addressed in future work of this thesis.

6.1 Discussion

In this section, we discuss topics of relevance to the thesis. These are as follows: the choice of testing techniques and coverage criteria, code translation, and selection of the Python programming language.

Choice of techniques and coverage criteria: Unit testing, in both manual and automated manners, is used in this thesis because of its popularity in the industry and its capability to enable a detailed view of the possible bugs in the code under test. Automated search-based testing equipped with a mutation analysis is a modern scientifically-proven testing technique used in this thesis. Requirement and branch coverage, as two of the de facto coverage criteria in the industry currently have also been used in this thesis wherever testing is applied. Thus, for coverage criteria measurement, this thesis attempts to research the merits and demerits of the selected test techniques, respectively,

in terms of requirement coverage.

Code Translation: A major contribution of this thesis is tied to the translation of described PLC programs in FBD/ST languages into Python code, to enable automated testing for PLCs. We acknowledge that translating a visual description of a PLC program, such as the one provided by FBD, into a dynamic text-based programming language such as Python, can be challenging since it demands considering several different aspects such as cyclic execution and non-existing data types, yet it is much less expensive than developing a whole new test generation tool for PLCs, from scratch. Hence, enabling mutation analysis and automated search-based testing via different meta-heuristic algorithms using an already available powerful tool such as Pynguin [38] is worth investigating.

Selection of the Python Programming Language: Choosing Python as the destination programming language in translating a PLC program in this thesis is justified by the following reasons. Firstly, Python is the only non-IEC61131-3 programming language that is supported by the IDE under focus in this study, that is, by CODESYS. Secondly, Python is equipped with several powerful testing and verification tools such as Pynguin [38] and Nagini [112], respectively, which enables a high level of flexibility and provides the basis for an increased level of assurance, when investigating the efficiency of the proposed translation framework in the context of PLC testing.

6.2 Limitations

One of the main limitations of this thesis concerns the generalizability of the results. We have successfully applied our proposed methods to a variety of different real-world PLC programs in the context of supervising port cranes and nuclear plant systems, however, the applicability and efficiency of the proposed methods in more sophisticated PLC programs need to be investigated to a larger extent.

Another limitation is the narrow field of investigation for choosing proper test automation tools for PLC programs. We have limited the scope of this investigation to the available test automation tools for CODESYS IDE because

of its popularity among industrial practitioners, yet these results cannot be generalized to other PLC IDEs. The other limitation of our work is connected to using a testing time budget when using the search-based algorithms in testing the PLC programs inside the PyLC translation framework in Papers B and D. We considered a 10 minutes upper bound time limit for testing the real-world PLC programs in paper B, while we increased it to 20 minutes in Paper D. increasing this time budget might affect the efficiency of Pynguin test automation tool in terms of automated search-based testing of PLC programs.

The final limitation is related to testing Timer blocks in the FBD programs under translation, using the proposed automated PLC to Python translation framework in this thesis. Automated PyLC is capable of simulating the behaviour of the timer function blocks in PLC programs, but when it comes to testing the translated program in Python using the Pynguin test generator, the testing tool is stuck in an infinite loop. This problem limits the testing capability of the PyLC tool in the context of Timer blocks in the proposed translation framework, however, a fix to this problem is under investigation and constitutes one of the directions of future work.

Chapter 7

Conclusion and Future Work

Motivated by the lack of automated techniques for testing PLC programs, this thesis provides methods and tools that enable test automation for described PLC programs in industrial settings. It proposes PyLC, an automated PLC to Python translation framework, and assesses the applicability of EARS semi-structured requirement engineering syntax [31] in PLC testing. This thesis evaluates the proposed methods of all included publications using real-world industrial control programs. The results of investigating the most-discussed test automation tools of CODESYS IDE among practitioners highlight that the CODESYS Test Manager and CoUnit as the prevalent test automation frameworks. This investigation implies key considerations for choosing a framework that includes cost, platform support, industrial applicability, and feature set. A comparison between the identified test automation tools of CODESYS IDE favours CODESYS Test Manager for its maturity, user support, and features such as record playback and test suite extension, although CoUnit, an open-source alternative, also offers significant functionality for PLC testing.

We introduce a proof of concept for the PyLC framework that translates a PLC program into Python code, and validates the correctness of the translated code through a hybrid 3-layered unit-testing validation mechanism. This thesis evaluates the applicability and efficiency of PyLC across various real-world industrial PLC programs. The results of this study imply that the manual version of PyLC is capable of translating a PLC program, described in FBD or ST languages, into executable Python code. The hybrid translation valida-

tion module of the PyLC framework has validated the translation correctness by achieving full coverage for requirement-based and translation-rules-based testing, followed by an average of 88.44% branch coverage for search-based testing.

As a continuation of the investigation of facilitating PLC testing for practitioners, we conduct an experiment on requirements engineering and testing for PLC systems, using the so-called EARS notation, which reveals that individuals employ various EARS patterns to transform identical requirements, indicating the flexibility of EARS syntax in formulating natural language requirements. This experiment is followed by a PLC testing investigation, which proposes a method for transforming a PLC requirement into an EARS requirement, and generating test cases out of it. The results of applying this method on different industrial real-world PLC programs in the context of crane supervision systems show that the EARS-generated requirement-based test cases are effective and provide a convenient way for PLC testers to articulate test specifications when compared to traditional methods.

In an attempt to improve the efficiency of the proposed PLC to Python translation framework, we also automate the PyLC framework by equipping it with an automated XML analyzer and an automated code generator, followed by an automated meta-heuristic translation validation module. The automated variation of PyLC is capable of translating a PLC program in FBD language as a PLCopen XML file into an executable Python code. The automated PyLC accomplishes the translation under the IEC 61131-3 standard and performs super fast without any manual human interventions. The results of applying the automated PyLC to different real-world PLC programs in the context of crane supervision and nuclear plant control systems reveal that the automated PyLC can translate various industrial FBD programs into Python code, with diverse block-type support. The automated translation validation module of PyLC, which is assisted by Pynguin [38], generates test cases effectively too, achieving an average branch coverage of 98% across ten real-world industrial PLC programs.

In future work, we plan to upgrade the proposed automated translation framework of this thesis, by adding support for PLC programs developed in the ST language. Furthermore, we will investigate the efficiency of using different search-based algorithms in testing PLC programs in an industrial setting, by employing the PyLC framework. To increase the accuracy and correctness

of the Python code obtained by translating PLC programs into Python, using the PyLC framework, we plan to equip PyLC with the static verifier for Python, called Nagini [112]. Developing an automated EARS-based test generator using PLC requirements in natural language is another future work direction of this thesis. Last but not least, we intend to perform a rigorous comparison between the efficiency of the proposed automated testing frameworks and the current manual PLC testing used in the industry.

.

Bibliography

[1] 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 International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 277–284. IEEE, 2022.

[2] 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 *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 1476–1485, 2023.

[3] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, and Cristina Seceleanu. An empirical investigation of requirements engineering and testing utilizing EARS notation in PLC programs. *Springer Nature Topical Issue on Advances in Combinatorial and Model-based Testing*, 2023.

[4] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Cristina Seceleanu, Wasif Afzal, and Filip Sebek. Automating test generation of industrial control software through a plc-to-python translation framework and pynguin. In *30th Asia Pacific Software Engineering Conference - APSEC 2023*. IEEE, 2023.

[5] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, and Cristina Seceleanu. An experiment in requirements engineering and testing using ears notation for plc systems. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 10–17. IEEE, 2023.

[6] Bela Genge. A review of model-based testing of industrial automation systems. *Journal of Systems Architecture*, 98:333–342, 2019.

[7] Sang-Gu Lee, Ji-Hoon Jang, Kyu-Chul Lee, and Moonzoo Kim. Model-based testing for automation software with improved code coverage. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 132–139. IEEE, 2017.

[8] Danial Habibi, José Antonio Macias, Héctor Ayala, and Pedro Sanchez. Fuzzing-based testing of industrial control systems: Challenges and opportunities. In *2020 IEEE/ACM 6th International Workshop on Security Testing (SECTEST)*, pages 28–33. IEEE, 2020.

[9] Héctor Ayala, Danial Habibi, José Antonio Macias, and Pedro Sanchez. Real-time simulation of large-scale power systems with hardware-in-the-loop. *IEEE Transactions on Industrial Informatics*, 15(2):1143–1152, 2019.

[10] Dong Cheng, Héctor Ayala, and Yuguang Huang. Intelligent control of renewable microgrid systems. *IEEE Transactions on Industrial Electronics*, 63(2):1117–1127, 2016.

[11] Rafael Blas, Francisco Moyano, Pedro Sánchez, and David Pérez. Plc program testing using model-based test case generation. *Computers in Industry*, 124:103368, 2021.

[12] David Havlik, Jiri Buresek, and Zdenek Riha. Model-based testing of cyber-physical systems: A case study on wind turbine control. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1144–1149. IEEE, 2019.

[13] Amr S. Mohamed, Hsiang Yen, and Robert X. Gao. A systematic review of security testing in industrial control systems. *IEEE Transactions on Industrial Informatics*, 16(8):5271–5282, 2020.

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

[15] Juergen Weber, Peter Feldmann, Roland Brandl, Stefan Almer, and Florian Matheis. Plc programming languages in the fourth industrial revolution. In *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 270–276. IEEE, 2018.

[16] Roland Brandl, Peter Feldmann, and Juergen Weber. Codesys development system–more than an iec 61131-3 programming tool. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2016.

[17] Vahid Garousi, Wasif Afzal, Adem Çağlar, İhsan Berk Işık, Berker Baydan, Seçkin Çaylak, Ahmet Zeki Boyraz, Burak Yolaçan, and Kadir Herkiloğlu. Comparing automated visual gui testing tools: an industrial case study. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, pages 21–28, 2017.

[18] Päivi Raulamo-Jurvanen, Mika Mäntylä, and Vahid Garousi. Choosing the right test automation tool: a grey literature review of practitioner sources. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 21–30, 2017.

[19] Marcin Jamro. Pou-oriented unit testing of iec 61131-3 control software. *IEEE Transactions on Industrial Informatics*, 11(5):1119–1129, 2015.

[20] Florian Hofer and Barbara Russo. Iec 61131-3 software testing: A portable solution for native applications. *IEEE Transactions on Industrial Informatics*, 16(6):3942–3951, 2019.

[21] Päivi Raulamo-Jurvanen, Simo Hosio, and Mika V Mäntylä. Practitioner evaluations on software testing tools. In *Proceedings of the Evaluation and Assessment on Software Engineering*, pages 57–66. 2019.

[22] Mirko Conrad. Testing-based translation validation of generated code in the context of iec 61508. *Formal Methods in System Design*, 35:389–401, 2009.

[23] 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.

[24] Klaus Schneider. The synchronous programming language quartz. Technical report, Internal Report 375, Department of Computer Science, University of Kaiserslautern, 2009.

[25] Marcel Christian Werner and Klaus Schneider. From iec 61131-3 function block diagrams to sequentially constructive statecharts. In *2022 Forum on Specification & Design Languages (FDL)*, pages 1–8. IEEE, 2022.

[26] Alistair Mavin Mav and Philip Wilkinson. Ten years of ears. *IEEE Software*, 36(5):10–14, 2019.

[27] Alistair Mavin, Philip Wilksinson, Sarah Gregory, and Eero Uusitalo. Listens learned (8 lessons learned applying ears). In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 276–282. IEEE, 2016.

[28] Mika V Mäntylä, Kai Petersen, Timo OA Lehtinen, and Casper Lassenius. Time pressure: a controlled experiment of test case development and requirements review. In *Proceedings of the 36th International Conference on Software Engineering*, pages 83–94, 2014.

[29] Fabiano Dalpiaz and Arnon Sturm. Conceptualizing requirements using user stories and use cases: a controlled experiment. In *Requirements Engineering: Foundation for Software Quality: 26th International Working Conference, REFSQ 2020, Pisa, Italy, March 24–27, 2020, Proceedings 26*, pages 221–238. Springer, 2020.

[30] Markus Weninger, Paul Grünbacher, Huihui Zhang, Tao Yue, and Shaukat Ali. Tool support for restricted use case specification: Findings from a controlled experiment. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 21–30. IEEE, 2018.

[31] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (ears). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322. IEEE, 2009.

[32] Meng Li, Baoluo Meng, Han Yu, Kit Siu, Michael Durling, Daniel Russell, Craig McMillan, Matthew Smith, Mark Stephens, and Scott Thomson. Requirements-based automated test generation for safety critical software. In *2019 38th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2019.

[33] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. Model-based test suite generation for function block diagrams using the uppaal model checker. In *2013 sixth international conference on software testing, verification and validation workshops*, pages 158–167. IEEE, 2013.

[34] Mehdi Malekzadeh and Raja Noor Ainon. An automatic test case generator for testing safety-critical software systems. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 1, pages 163–167. IEEE, 2010.

[35] TJ Prati, JM Farines, and MH De Queiroz. Automatic test of safety specifications for plc programs in the oil and gas industry. *IFAC-PapersOnLine*, 48(6):27–32, 2015.

[36] Havva Gulay Gurbuz and Bedir Tekinerdogan. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal*, 26:1327–1372, 2018.

[37] Emil Alégroth, Kristian Karl, Helena Rosshagen, Tomas Helmfridsson, and Nils Olsson. Practitioners' best practices to adopt, use or abandon model-based testing with graphical models for software-intensive systems. *Empirical Software Engineering*, 27(5):1–42, 2022.

[38] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[39] Bart Broekman and Edwin Notenboom. *Testing embedded software*. Pearson Education, 2003.

[40] Alistair Sutcliffe. *Requirements Analysis for Safety Critical Systems*, pages 149–180. Springer London, London, 2002.

[41] Jan L Rouvroye and Elly G van den Bliek. Comparing safety analysis techniques. *Reliability Engineering & System Safety*, 75(3):289–294, 2002.

[42] M. Maslar. PLC standard programming languages: IEC 1131-3. In *Conference Record of 1996 Annual Pulp and Paper Industry Technical Conference*, pages 26–31, 1996.

[43] Iris Graessler and Julian Hentze. The new V-Model of VDI 2206 and its validation. *at-Automatisierungstechnik*, 68(5):312–324, 2020.

[44] Bohan Liu, He Zhang, and Saichun Zhu. An incremental V-model process for automotive development. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 225–232. IEEE, 2016.

[45] Liu Shuping and Pang Ling. The research of V model in testing embedded software. In *2008 International Conference on Computer Science and Information Technology*, pages 463–466. IEEE, 2008.

[46] Ravi Shanker Yadav. Improvement in the V-Model. *International Journal of Scientific & Engineering Research*, 3(2):1–6, 2012.

[47] Márcio Eduardo Delamaro, José Carlos Maldonado, and Márcio Jino. Mutation testing: a technique for assessing the quality of your test cases. *Software, IEEE*, 18(4):76–82, 2001.

[48] Jeffrey M Voas. Automating boundary testing of real-number programs. *Software Engineering, IEEE Transactions on*, 23(6):337–348, 1997.

[49] Elsye Ginting. Equivalence partitioning and boundary value analysis to test information system: case study. *International Journal of Scientific & Engineering Research*, 6(1):164–170, 2015.

[50] Michael Hutchins, Harry Foster, Tushar Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and control-flow-based test adequacy criteria. *Software Engineering, IEEE Transactions on*, 20(8):697–708, 1994.

[51] C Mayer, A Sillitti, and G Succi. Manual vs. automated testing: Which one is better for you? *Journal of Systems and Software*, 168:110610, 2020.

[52] Reinhard Hametner, Ingo Hegny, and Alois Zoitl. A unit-test framework for event-driven control components modeled in iec 61499. In *Proceedings of the 2014 ieee emerging technology and factory automation (etfa)*, pages 1–8. IEEE, 2014.

[53] Dietmar Winkler, Reinhard Hametner, and Stefan Biffl. Automation component aspects for efficient unit testing. In *2009 IEEE Conference on Emerging Technologies & Factory Automation*, pages 1–8. IEEE, 2009.

[54] Dong Li, Linghuan Hu, Ruizhi Gao, W Eric Wong, D Richard Kuhn, and Raghu N Kacker. Improving MC/DC and fault detection strength using combinatorial testing. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 297–303. IEEE, 2017.

[55] Gunwant Dhadyalla, Neelu Kumari, and Timothy Snell. Combinatorial testing for an automotive hybrid electric vehicle control system: a case study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 51–57. IEEE, 2014.

[56] A Rengarajan and Raja Praveen KN. An innovation of differential unit tests based smart industrial automation software debugging tool. In *2023 International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE)*, pages 1–6. IEEE, 2023.

[57] Rob Palin, David Ward, Ibrahim Habli, and Roger Rivett. ISO 26262 safety cases: Compliance and assurance. 2011.

[58] Ron Bell. Introduction to iec 61508. In *Acm international conference proceeding series*, volume 162, pages 3–12. Citeseer, 2006.

[59] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 67–77, 2012.

[60] Miriam Ugarte Querejeta, Eunkyoung Jee, Lingjun Liu, Pablo Valle, Aitor Arrieta, and Miren Illarramendi Rezabal. Search-based test case

selection for plc systems using functional block diagram programs. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 228–239. IEEE, 2023.

[61] Kivanc Doganay, Markus Bohlin, and Ola Sellin. Search based testing of embedded systems implemented in iec 61131-3: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 425–432. IEEE, 2013.

[62] Susanne Kandl, Raimund Kirner, and Peter Puschner. Development of a framework for automated systematic testing of safety-critical embedded systems. In *2006 International Workshop on Intelligent Solutions in Embedded Systems*, pages 1–13. IEEE, 2006.

[63] Paul C Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013.

[64] Igor Buzhinsky, Vladimir Ulyantsev, Jari Veijalainen, and Valeriy Vyatkin. Evolutionary approach to coverage testing of iec 61499 function block applications. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1213–1218. IEEE, 2015.

[65] Eunkyoung Jee, Junbeom Yoo, Sungdeok Cha, and Doohwan Bae. A data flow-based structural testing technique for fbd programs. *Information and Software Technology*, 51(7):1131–1139, 2009.

[66] Marco Lormans, Hans-Gerhard Gross, Arie Van Deursen, Rini Van Solingen, and André Stehouwer. Monitoring requirements coverage using reconstructed views: An industrial case study. In *2006 13th Working Conference on Reverse Engineering*, pages 275–284. IEEE, 2006.

[67] RVRK Kavitha, VR Kavitha, and N Suresh Kumar. Requirement based test case prioritization. In *2010 International Conference on Communication Control and Computing Technologies*, pages 826–829. IEEE, 2010.

[68] Anne Kramer and Bruno Legeard. *Model-based testing essentials-guide to the ISTQB certified model-based tester: foundation level*. John Wiley & Sons, 2016.

[69] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[70] Giovanni Grano, Timofey V Titov, Sebastiano Panichella, and Harald C Gall. Branch coverage prediction in automated testing. *Journal of Software: Evolution and Process*, 31(9):e2158, 2019.

[71] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, pages 194–212, 2012.

[72] Ahmed Gario, Anneliese Andrews, and Seana Hagerman. Testing of safety-critical systems: An aerospace launch application. In *2014 IEEE Aerospace Conference*, pages 1–17. IEEE, 2014.

[73] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *2017 39th International Conference on Software Engineering (ICSE)*, pages 597–608. IEEE, 2017.

[74] Gregory Gay, Matt Staats, Michael W Whalen, and Mats PE Heimdahl. Moving the goalposts: coverage satisfaction is not enough. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, pages 19–22, 2014.

[75] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014.

[76] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation analysis. Technical report, Georgia Inst of Tech Atlanta School of Information And Computer Science, 1979.

[77] Junbeom Yoo, Eui-Sub Kim, and Jang-Soo Lee. A behavior-preserving translation from fbd design to c implementation for reactor protection

system software. *Nuclear Engineering and Technology*, 45(4):489–504, 2013.

[78] Eunkyoung Jee, Donghwan Shin, Sungdeok Cha, Jang-Soo Lee, and Doo-Hwan Bae. Automated test case generation for fbd programs implementing reactor protection system software. *Software Testing, Verification and Reliability*, 24(8):608–628, 2014.

[79] 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.

[80] Felix Dobslaw, Ruiyuan Wan, and Yuechan Hao. Generic and industrial scale many-criteria regression test selection. *Journal of Systems and Software*, 205:111802, 2023.

[81] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven validation & verification environment for embedded systems. In *2008 International Symposium on Industrial Embedded Systems*, pages 241–244. IEEE, 2008.

[82] Marcel Christian Werner and Klaus Schneider. From iec 61131-3 function block diagrams to sequentially constructive statecharts.

[83] Birgit Vogel-Heuser, Juliane Fischer, Stefan Feldmann, Sebastian Ulewicz, and Susanne Rösch. Modularity and architecture of plc-based software for automated production systems: An analysis in industrial companies. *Journal of Systems and Software*, 131:35–62, 2017.

[84] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6:239–242, 2014.

[85] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42. IEEE, 2012.

[86] Eduard Enoiu, Daniel Sundmark, Adnan Čaušević, and Paul Pettersson. A comparative study of manual and automated testing for industrial control software. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 412–417. IEEE, 2017.

[87] Stefan Berner, Roland Weber, and Rudolf K Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579, 2005.

[88] Vahid Garousi and Frank Elberzhager. Test automation: not just for test execution. *IEEE Software*, 34(2):90–96, 2017.

[89] Colin Robson and Kieran McCartan. *Real world research: a resource for users of social research methods in applied settings*. Wiley, 2016.

[90] Conradi and Hofmann. *Empirical Methods and Studies in Software Engineering*. Springer Berlin Heidelberg, 2003.

[91] Robert K. Yin. *Case study research and applications: Design and methods*. Sage Publications, 2018.

[92] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[93] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin, 2002.

[94] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[95] John W. Creswell and J. David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage Publications, 2017.

[96] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proceedings of the 20th international conference on evaluation and assessment in software engineering*, pages 1–6, 2016.

[97] 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 Engineering*, 44(2):122–158, 2017.

[98] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. In *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*, pages 3–17. Springer, 2017.

[99] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007.

[100] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[101] Alessio Ferrari, Franco Mazzanti, Davide Basile, and Maurice Ter Beek. Systematic evaluation and usability analysis of formal methods tools for railway signaling system design. *IEEE Transactions on Software Engineering*, 2021.

[102] Mubarak Albarka Umar and Chen Zhanfang. A study of automated software testing: Automation tools and frameworks. *International Journal of Computer Science Engineering (IJCSE)*, 6:217–225, 2019.

[103] Heidilyn Veloso Gamido and Marlon Viray Gamido. Comparative review of the features of automated software testing tools. *International Journal of Electrical and Computer Engineering*, 9(5):4473, 2019.

[104] Harpreet Kaur and Gagan Gupta. Comparative study of automated testing tools: selenium, quick test professional and testcomplete. *Journal of Engineering Research and Applications*, 3(5):1739–1743, 2013.

[105] Emil Borjesson and Robert Feldt. Automated system testing using visual gui testing tools: A comparative study in industry. In *International Conference on Software Testing, Verification and Validation*, pages 350–359. IEEE, 2012.

[106] Sebastian Ulewicz and Birgit Vogel-Heuser. Increasing system test coverage in production automation systems. *Control Engineering Practice*, 73:171–185, 2018.

[107] Sebastian Ulewicz and Birgit Vogel-Heuser. Guided semi-automatic system testing in factory automation. In *International Conference on Industrial Informatics (INDIN)*, pages 142–147. IEEE, 2016.

[108] Sebastian Ulewicz and Birgit Vogel-Heuser. System regression test prioritization in factory automation: Relating functional system tests to the tested code using field data. In *Annual Conference of the IEEE Industrial Electronics Society*, pages 4619–4626. IEEE, 2016.

[109] Sebastian Ulewicz and Birgit Vogel-Heuser. Industrially applicable system regression test prioritization in production automation. *Transactions on Automation Science and Engineering*, 15(4):1839–1851, 2018.

[110] GIACOMO Barbieri, GABRIEL Quintero, OSCAR Cerrato, JULIAN Otero, DAVID Zanger, and ALEJANDRO Mejia. A mathematical model to enable the virtual commissioning simulation of wick soilless cultivations. *J. Eng. Sci. Technol*, 16:3325–3342, 2021.

[111] Flemström Daniel, Enoiu Eduard, Azal Wasif, Sundmark Daniel, Gustafsson Thomas, and Kobetski Avenir. From natural language requirements to passive test cases using guarded assertions. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 470–481. IEEE, 2018.

[112] Marco Eilers and Peter Müller. Nagini: a static verifier for python. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, pages 596–603. Springer, 2018.

II

Included Papers

Chapter 8

Paper A: Choosing a Test Automation Framework for Programmable Logic Controllers in CODESYS Development Environment

Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu
Published in the 15th IEEE International Conference on IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2022), The Next Level of Test Automation (NEXTA 2022).

8.1 Abstract

Programmable Logic Controllers are computer devices often used in industrial control systems as primary components that provide operational control and monitoring. The software running on these controllers is usually programmed in an Integrated Development Environment using a graphical or textual language defined in the IEC 61131-3 standard. Although traditionally, engineers have tested programmable logic controllers' software manually, test automation is being adopted during development in various compliant development environments. However, recent studies indicate that choosing a suitable test automation framework is not trivial and hinders industrial applicability. In this paper, we tackle the problem of choosing a test automation framework for testing programmable logic controllers, by focusing on the COntroller DEvelopment SYStem (CODESYS) development environment. CODESYS is deemed popular for device-independent programming according to IEC 61131-3. We explore the CODESYS-supported test automation frameworks through a grey literature review and identify the essential criteria for choosing such a test automation framework. We validate these criteria with an industry practitioner and compare the resulting test automation frameworks in an industrial case study. Next, we summarize the steps for selecting a test automation framework and the identification of 29 different criteria for test automation framework evaluation. This study shows that CODESYS Test Manager and CoUnit are mentioned the most in the grey literature review results. The industrial case study aims to increase the know-how in automated testing of programmable logic controllers and help other researchers and practitioners identify the right framework for test automation in an industrial context.

8.2 Introduction

Testing is an important activity in the engineering of industrial control software. In certain application domains (e.g., automation industry), programmable logic controllers (PLCs) provide management and monitoring for control software [1]. Even if test execution on PLCs is usually performed manually, test automation is emerging during PLC development at different stages of integration. Different PLC vendors and PLC software manufacturers have proposed several Integrated Development Environments (IDEs). One of the

most frequently used PLC IDEs in the industry is CODESYS, a manufacturer-independent software that is free to use. It supports all the PLC programming languages of IEC 61131-3 standard and is widely used by many industrial companies.

Test automation can be defined as the process of automating software testing tasks such as test script development, test execution, and requirements verification using an automation test framework [2], [3]. Choosing the right test automation tool has received significant attention from both academia and industry in recent years[4]. Furthermore, recent observations of collaborations between industry and academia emphasize the importance of selecting the right test automation tool since it is a non-trivial task for many practitioners [5]. This could stem from at least two reasons; the misunderstanding of what important criteria to use for choosing the right tool and the lack of knowledge of the pros and cons of using test automation frameworks in practice.

In this paper, we address the problem of choosing the right test automation tool for PLC programs in CODESYS IDE by leveraging a Grey Literature Review (GLR) followed by a comparative study on the discovered tools. Aiming at conducting an effective comparison between the detected tools, we discover the most important features of the test automation tools through a literature review. We evaluate the validity of the discovered features by asking a group of engineers from a large automation company to review them. Based on the goal of this study, we formulate the following research questions:

1. What are the reported test automation frameworks for CODESYS in the grey literature?

2. What are the reported features that should be considered when choosing a test automation framework for CODESYS?

3. How do different test automation frameworks for CODESYS compare in terms of different features?

4. How do different test automation frameworks for CODESYS compare in terms of their applicability to an industrial use case?

We aim to answer these research questions using a GLR and a case study in which we compare the identified test automation frameworks.

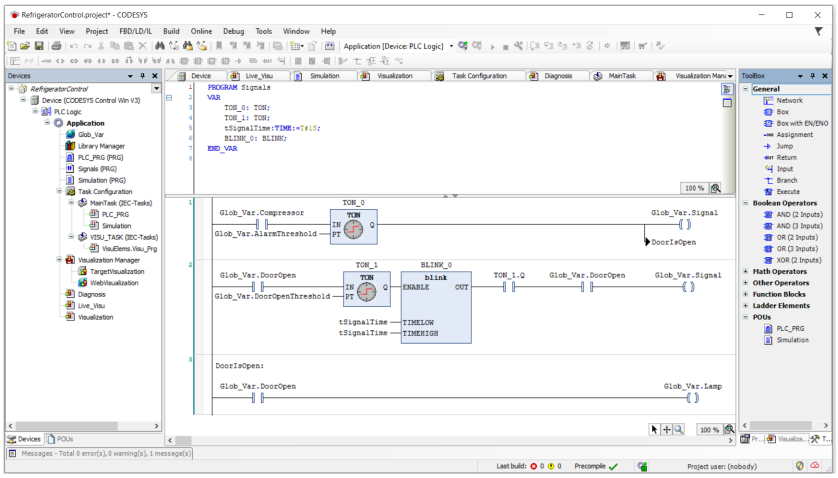

Figure 8.1: CODESYS Integrated Development Environment

8.3 BACKGROUND AND RELATED WORK

8.3.1 PLC Programming, IEC 61131-3 and CODESYS

In recent years the IEC 61131-3 programming standard for the automation industry has been proposed, and today it is widely accepted and used by a variety of well-known PLC manufacturers worldwide. The smallest independent software unit in a PLC program is called a POU (Program Organisation Unit), also known as a block. There are three types of POU: *function*, *function block* (FB), and *program*, which can call each other with or without parameters. Based on IEC 61131-3 standard, a POU can be programmed using several programming languages [6] (i.e., *structured text* (ST), *instruction list* (IL), *ladder diagram* (LD), *Sequential Function Chart* (SFC) and *function block diagram* (FBD)).

CODESYS stands for COntroller DEvelopment SYStem, and it is an integrated development environment (IDE) for programming controller applications according to the international industrial standard IEC 61131-3 [7]. The framework is developed by Smart Software Solutions GmbH In this work, we choose CODESYS as our IDE for two reasons. First, CODESYS is free to use and is popular in industry [7]. Second, CODESYS is a device-manufacturer-

independent IDE [1] that can be used to develop PLC programs for a wide range of PLC devices from various vendors.

8.3.2 Related Work

In recent years, researchers have made efforts to develop test automation frameworks for PLC software. Jamro introduces a method for POU-oriented unit testing for IEC 61131-3 languages [8]. In this approach, test cases are defined in CPTest+, a dedicated test definition language. The proposed approach is introduced in the CPDev engineering environment. Recently, Hofer and Russo [9] presented a unit-testing framework named APTest (Advanced Program Organization Unit Testing) for CODESYS IDE. The framework is developed based on the IEC61131-3 standard and CPTest+. APTest is a POU-based framework equipped with a test library supporting different types of assertions and is compatible with CODESYS (version 2.3). Even if these academic tools have a wide range of capabilities such as test parallelization, simulating analogue signals, and supporting time-dependent behaviours, there is limited evidence of how industrially useful these frameworks are. In addition, these tools are only compatible with older versions of CODESYS.

Selecting a test automation framework is an essential part of software testing, and recent studies have looked at different challenges to implementing automation support. Raulamo-Jurvane et al. [5] performed a GLR to identify the practitioners' criteria for choosing the right test automation tools. The study showed that practitioners select and embrace the widely known and utilized tools. Garousi et al. [4] compared visual GUI testing frameworks (i.e., Sikuli and JAutomate) using several relevant features and performed an industrial case study. In 2019, Raulamo-Jurvane et al. investigated the practitioners' opinions on evaluating testing tools by conducting an online survey [10]. They found that evaluations in which one uses a tool seem to be more favourable than those based on opinions and considering the opinions of seven experts provides a reasonable level of reliability.

These results kindled our interest in studying how to tackle the problem of choosing a test automation framework for programmable logic controllers in CODESYS, especially when these tools are used to test safety-critical industrial control systems.

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

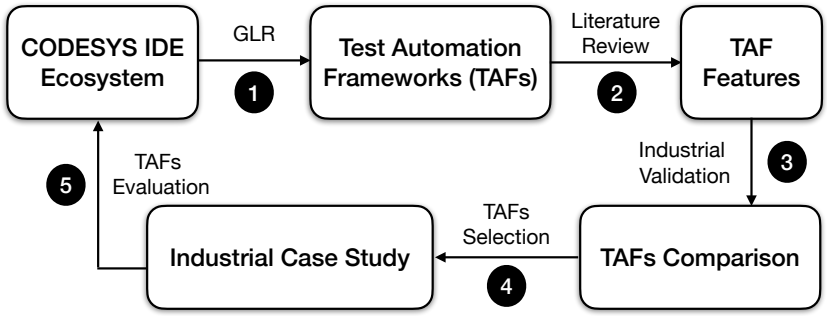

Figure 8.2: An Overview of The Methodology Used for Choosing Test Automation Frameworks (TAFs) for PLCs.

8.4 METHOD

In this study, we leverage a hybrid methodology that combines conducting a GLR and an industrial case study. Our aim of performing GLR is to find the most-discussed available test automation frameworks of CODESYS IDE systematically and reasonably, reflecting the practitioners' point of view. In contrast, the case study is performed to represent and compare the functionality of the proposed frameworks in a real-world industrial case study. The overall conceptual architecture of the proposed hybrid methodology can be observed in Figure 8.2, and it will be discussed in more detail in the rest of this section.

8.4.1 Grey Literature Review

We find the GLR as one of the most suitable approaches to conduct our study since the available information about test automation frameworks of CODESYS is more accessible online than in academic papers. Consequently, to explore the available automation frameworks targeting CODESYS, we need to gather information across the web, and prioritize it based on features and popularity. We base our approach on the GLR approach proposed by Garousi e al. [11]. The conducted GLR approach (Step 1 in Figure 8.2) consists of several steps.

Table 8.1: GLR Search Strings

A	B	C	D
CODESYS	Test	Automation	Framework
	Testing	Automated	Tool
		Automatic	

Table 8.2: Considered Object Properties

Object Properties												
Title	Link	Extracted framework	Object Type					Literature Type		Demographic info		
			Documentation	Tutorial	Description	Discussion	other	Grey	Formal	Year	Author(s)	Author's Organisation

8.4.2 Search Process and Framework Selection

First, we aim to detect the test automation frameworks available for CODESYS, and consequently, we perform a GLR by searching for a combination of topic-related keywords on Google. We carry out several exploratory search queries using strings such as "CODESYS test automation framework" and "CODESYS test automation tool". Moreover, we also consider the related search strings suggested to us by Google. However, we remove the suggestions that include a specific framework name. We present the final search strings in Table 1. We consider the keywords of each column as synonyms and combine them via the OR operator (e.g., framework OR tool). Then, we produce the whole search string using the AND operator between the existing keywords of columns A to D.

We follow the guidelines used by Raulamo et al. [5] to identify the available test automation frameworks targeting CODESYS. Firstly, the pool of contents is revised by the first author of this paper to remove any irrelevant results from it. Secondly, the new version of the pool is reviewed by at least one of the authors of this paper. We only include the results related to CODESYS-

Table 8.3: Selection Criteria for GLR

Selection Criteria			
Author's Credibility		CODESYS Verified	v3.x Support
Alexa Rank	Referral Sites		

compatible test automation frameworks. Furthermore, all the academic papers are excluded from the contents pool. It should be noted that in case of any conflicting views among authors, we leverage a voting system to choose the results for the final pool, based on the opinion of the majority of the authors.

8.4.3 Pool of Objects

Every result and its corresponding information is called an "Object" in the rest of this study. We consider several properties for every object. The considered properties are shown in Table 8.2. The final pool of objects and defined criteria are accessible online on GitHub[2].

8.4.4 Data Extraction Method

To classify the objects of the pool of contents and specify the required criteria to detect the efficient test automation frameworks of CODESYS, we follow a systematic qualitative method of specific related work [12]. All authors of this paper have reviewed the qualitative analysis of this work. Since coding in qualitative analysis is not just a preliminary step of analyzing the data, but it also includes "deep analysis and interpretation of the data meanings" [12], we select the relevant results and analyze them.

8.4.5 Selection Criteria

The final version of the considered criteria to classify the objects of the pool is shown in Table 8.3. Since the goal is to discover valid test automation frameworks that are compatible with CODESYS IDE, we checked the official CODESYS documentation. The next criterion is the credibility of the object's author, for which two parameters are evaluated: the number of referral sites to an object, and the Alexa rank of the object's website. Moreover, as we are looking for up-to-date frameworks compatible with the CODESYS v3.x family, the publishing date is another criterion used in filtering the results. The first version of the CODESYS V3.x family was released in 2016[3].

[2] https://github.com/MikaelSalari/CODESYS-Tool-Comparison
[3] https://store.codesys.com/en/codesys.html

8.4.6 Discovery and Validation of Features

To perform an effective comparison between the discovered test automation frameworks of CODESYS, first, we need to identify the most important reported features of test automation frameworks. To this end, we conduct an informal literature review on the related work in this area to gather a list of features that should be considered during the comparison process (Step 2 in Figure 8.2).

In this paper, we reviewed the literature and identified five existing works as our sources of information for test automation framework feature extraction: (1) The study of Ferrari et al. [13] on the selection and adoption of formal tools in the railway domain, (2) Umar et al. [14] which is an overview of popular existing test automation tools, (3) the study of Gamido et al. [15] performing a comparative review based on the user's needs, (4) a study [16] focusing on some well-known test automation frameworks including Selenium, Quick Test Professional and *Testcomplete*, and (5) a comparative study [17] on two different automated visual GUI testing tools including *CommercialTool* and *Sikuli*.

Aiming at making this work more aligned with industry, we asked a group of engineers who are working at a large industrial automation company in Sweden to evaluate the validity of the discovered features from their point of view (Step 3 in Figure 8.2). These engineers are experts in developing and testing the supervisory PLC programs. Besides feature validation, the engineer also added two new features that are important from the company's perspective in choosing the right test automation tool. We only include the industrial-validated features in our tool comparison study, since our priority is to help practitioners in their choice, based on their needs.

8.4.7 Industrial Case Study

To practically use the results of this comparison, we evaluate the applicability of the discovered test automation frameworks for CODESYS in a real-world scenario, using an exploratory case study using an industrial system (Step 4 & 5 in Figure 8.2). The case is provided by a large industrial automation company in Sweden.

8.5 Results

8.5.1 RQ1 - Discovered Test Automation Frameworks

As a result of the GLR, we obtained 120000 search results which are all written in English. We only stored the first 100 results locally to build the pool of contents since we discovered that these contain relevant sources to our topic. Most of the objects in the final version in the pool of objects have been published by industry individuals, including IDE developers and PLC vendors. Aiming to establish a trade-off between the preferences of companies and independent framework developers in our results, we included valid third-party developers and GitHub topics in the final pool. After reviewing the content of the pool, we ended up with a pool consisting of 13 sources. After analyzing the final objects based on the defined criteria, we discovered three test automation frameworks as the most prevalent automation frameworks targeting CODESYS. Out of all the collected results, 62% of the objects in the pool are pointing towards CODESYS Test Manager[4] (the largest share of the discovered objects). Two other frameworks, CoUnit[5] and TcUnit are revealed in 15% of the objects each. Other frameworks were mentioned in 8% of the objects. Our results suggest that most of the discovered objects of our GLR after screening and applying the selection criteria point towards CODESYS Test Manager, CoUnit (formerly known as CfUnit), and TcUnit as the predominant test automation frameworks targeting CODESYS. We note here that CoUnit is developed based on TcUnit and both frameworks have similar functionality. Since CODESYS IDE officially supports only the former, we include CoUnit in the final list of discovered automation frameworks.

Answer RQ1: Our results suggest that the most prevalent test automation frameworks targeting CODESYS IDE for PLC testing are the CODESYS Test Manager and CoUnit.

8.5.2 RQ2 - Test Automation Frameworks Features

Conducting a comparison between the discovered test automation frameworks of CODESYS, first, we need to identify the essential features of these test au-

[4] https://store.codesys.com/codesys-test-manager.html?
[5] https://forge.codesys.com/lib/counit/home/Home/

Table 8.4: Extracted and Validated Framework Features.

Industry-validated Features		
Category	**Feature**	**Extraction Source**
Company Constraints	Cost	[13], [14], [15], [16], [17]
	Supported Platforms	[13], [14],[15]
Maturity	Industrial Usage	[13]
	Stage of Development	[13]
Testing Functionalities	Documentation and Report Generation	[13]
	Playback Record	[15]
	Test Suite Support	[17]
	Test Suite Extension	Industry
Tool Flexibility	Teamwork Support	[13]
Usability	DevOps/ALM Integration Support	[14]
	Continuous Integration (CI) Support	[14]
	Script Language	[14], [15], [17]
	Availability of Customer Support	[13], [14]
	Quality of Documentation	[13]
	Maintenance Support	Industry

tomation frameworks. To this end, we followed a hybrid approach which consisted of a literature review of related works followed by an industrial feature validation. Based on three sources of information used (academic works, industrial input, and official documentation), we discovered 29 industry-reported essential features that should be considered when choosing a test automation framework for PLCs. We acknowledge that many of these features are generic. Still, the instantiation of these features is specific to PLCs. Since our aim in conducting this work is to address the needs of industrial practitioners, we evaluated the validity of the discovered features by checking them with a group of engineers working with CODESYS and PLC testing in an industrial automation company in Sweden. These engineers validated these features of a test automation framework by marking the ones a tester would use to choose such a framework (i.e., 15 out of 29 features were considered important by these engineers). The list of the discovered and validated test automation framework features and non-validated ones as well as their category and source of extraction, are shown in Table 8.4 and 8.5 respectively. It should be noted that the gathered data does not need any further processing (e.g., open coding).

We divided the discovered features into five categories based on their focus, including Company Constraints, Maturity, Testing Functionalities, Framework

Table 8.5: Other Extracted Framework Features.

Other Features		
Category	**Feature**	**Extraction Source**
Company Constraints	Ease of Installation	[13], [14]
	License Type	Tool Documentation
Testing Functionalities	Test Script Specification	[13]
	Supported Testable Objects	Tool Documentation
	Requirements Traceability	[13]
	Script Creation Time	[14]
	Import Support	[17]
Tool Flexibility	Backward Compatibility	[13], [17]
	Standard Input Format	[13]
	Modularity of The Tool	[13]
	Framework Development Language	[17]
Usability	Programming skills	[14], [15]
	Report Format	[15]
	Graphical User Interface (GUI)	[13]

Flexibility, and Usability.

Company Constraints

Cost indicates the cost model used (FREE, MIX, PAY) [13], **Supported Platforms** specifies the platforms supported by the framework (Windows, Mac, Linux) [13], **Ease of Installation** indicates whether the framework installation requires installing other additional components or covers all the installation requirements. (YES, NO, PARTIAL) [13], and **License Type** implies the type of license used for the test automation framework (e.g., Apache, MIT).

Maturity

Industrial Usage specifies the level of reported industrial usage in academic papers and reports (HIGH, MEDIUM, LOW) [13] and **Stage of Development** indicates whether the framework has evolved through releasing different versions (MATURE), it is an academic or early version (PROTOTYPE) or it is new but has strong fundamental roots (PARTIAL) [13].

Testing Functionalities

Test Script Specification determines how the test script is represented by the framework: Graphical User Interface (GUI), Textual Representation (TEXT), imported textual file (IMPORT) [13]. **Supported Testable Objects** feature was discovered by reviewing the tools documentation. This feature indicates the object types that are supported for testing in a PLC program (APPLICATION, IEC LIBRARIES, COMMUNICATION). **Documentation and Report Generation** characterizes whether the automatically generated reports and documentation of a tool contain well-detailed technical details (COMPLETE) or only some summarized technical details are available (SUMMARIZED) [13]. **Requirements Traceability** specifies if the framework can provide traceability between the generated test cases to other related artefacts (YES, NO) [13]. **Script Creation Time** relates to the time required to produce test scripts (QUICK, SLOW) [14]. **Playback Record** indicates whether the framework can record testing sessions and playback these as test scripts (YES, NO) [15]. **Import Support** specifies if the framework can import test cases and test scripts using external files (Python, Java, NO) [17]. **Test Suite Support** relates to the framework's ability to support the user in the creation and execution of test suites (YES, NO) [17]. **Test Suite Extension** indicates the ease of extending test suites using the provided features of a certain test automation tool. Developing new test suites in a PLC program is a crucial and sensitive task because all the connections between the different test suites (test counterparts of a POU under test) should be updated after any new modifications. (EASY, MEDIUM, HARD).

Framework Flexibility

Backward Compatibility indicates to which extent test scripts developed with previous versions of the CODESYS framework can be used in the current version (YES, NO, UNCERTAIN) [13]. **Standard Input Format** specifies whether the language that is used for developing test cases is based on a standardized programming language or not (YES, NO) [13]. **Modularity of The Tool** specifies if the framework supports a wide range of different modules and add-ons that can be used to extend its functionality or not (YES, NO) [13]. **Teamwork Support** indicates whether the framework supports multi-user development and collaboration (YES, NO) [13]. **Framework Develop-**

ment Language details the programming language that is used to develop the framework (e.g., Python, Java, C, Jython) [17].

Usability

Programming Skills specifies what level of programming skills is needed to work with the framework. Available options are advanced needed programming skills (ADVANCED), no programming skills required (NOT REQUIRED) or only required for advanced test scripts (PARTIAL) [14]. **DevOps/ALM Integration Support** relates to the framework's ability to support integration with DevOps or ALM environments (YES, NO) [14]. **Continuous Integration (CI) Support** indicates if the framework supports CI frameworks (YES, NO) [14]. **Script Language** specifies the programming language(s) required for creating test scripts (e.g., Python, Structured Text, Function Block Diagram, Ladder) [14]. **Report Format** indicates how the test reports are represented in a framework (HTML, XML, CSV) [15]. **Availability of Customer Support** evaluates the level of support and tutorials provided for the users of a certain tool (HIGH, MEDIUM, LOW) [13]. **Graphical User Interface (GUI)** investigates the suitability of the designed graphical user interface of a framework. Available options are; The GUI is designed properly and is powerful enough to cover almost all the available functionalities of a framework (YES); The provided GUI is user-friendly, but does not provide a graphical representation of all functionalities of a framework (PARTIAL); The GUI exists but it is limited in its functionality (LIMITED); the framework has no GUI (NO) [13]. **Quality of Documentation** Specifies the framework's level of documentation and tutorial which is provided by the framework developers. Available options are; The framework is well-documented and a wide range of updated tutorials and framework specifications are easily accessible online (VERY GOOD); The framework is documented properly but the provided documentation is not easily accessible or it is only available offline (GOOD); The framework is not documented sufficiently or it can not be accessed easily (INSUFFICIENT) [13]. **Maintenance Support** implies the level of maintenance support that is provided and whether testing new functions in a POU under test is easy or not (EASY/HARD).

Results RQ2: We discovered several features that should be considered when choosing a test automation framework for PLC testing: cost, supported platforms, industrial use, stage of development, documentation and report generation, record playback, test suite support, test suite extension, team support, DevOps/ALM support, continuous integration support, scripting language, import support, availability of customer support, quality of documentation, and maintenance support.

8.5.3 RQ3 - Test Automation Frameworks

We conducted an initial comparative examination given the features identified in the previous section. We focus on Test Manager and CoUnit as our chosen test automation frameworks in CODESYS IDE. The results of this comparative examination are shown in Table 8.6. Even if both frameworks support only Windows platforms and have continuous integration support, we can observe significant differences. In terms of cost, CODESYS Test Manager is a commercial product but available for academics to use in their research. CoUnit is an open-source software freely available. Industrial usage of Test Manager is considered HIGH since its use has been reported in several industry-related reports [18], [19], [20], [21], [22]. Regarding the framework's maturity, CODESYS Test Manager seems to be more mature and has evolved through eight different versions so far, compared to CoUnit (i.e., in 3 versions). One of the main advantages of CODESYS Test Manager is the ability to record and playback that is not supported by the counterpart framework. Both frameworks support test suites in .xml file format. In addition, CODESYS Test Manager has the advantage of supporting .tsd (Tamino Schema) extension (used as a container of elements that a Tamino XML Server document contains). CODESYS Test Manager supports test suites to be extended by using specific predefined test commands, but the extension of test suites in CoUnit demands ST programming knowledge, and the user needs to instantiate the code for each single test case. CODESYS Test Manager supports Python and all IEC 61131-3 programming languages for developing the test scripts, while CoUnit only supports the ST programming language. Availability of customer support is another essential factor from an industry point of view in this comparison, and the CODESYS Test Manager seems to be superior in this respect. The quality of the documentation provided by the CODESYS Test Manager is excellent since

comprehensive educational material and good video tutorials are available. On the other hand, CoUnit provides less documentation and tutorials. Maintenance support is another important feature proposed. CODESYS Test Manager supports direct main PLC program testing and one instantiation of the code under test can be used in all related test suites but these features are not available in CoUnit.

Results RQ3: Based on our initial comparison between CODESYS Test Manager and CoUnit based on the 15 industry-validated features, the results show that CODESYS Test Manager is more mature and has several advantages over CoUnit, including user support, record and playback features, and easy test suite extension. Nevertheless, CoUnit, as an open-source counterpart, also provides testers with many key features used during PLC testing.

8.5.4 RQ4 - Applicability in an Industrial Case Study

Aiming to answer this research question, we applied the two test automation frameworks we found through our GLR to an industrial case study by considering several possible test scenarios. Our case is a control system provided by a large automation company in Sweden consisting of several POUs. This system is developed in the FBD programming language.

The Function Block (FB) in this POU consists of several computational blocks executed cyclically. The program executes in a cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs). The FBD program is created as a composition of interconnected blocks with data flow communication. When activated, a program consumes one set of input data and then executes it to completion. We considered functional scenarios for testing the POU. We evaluate this functionality and the applicability of *Test Manager* and *CoUnit* by automating the test execution for the provided case and all POUs. To this end, we generated several test suites consisting of manually created test cases.

To automate the test execution in CODESYS Test Manager, the first step needed is instantiating the POU in the main PLC program. Next, we create the required test suites containing 10 test cases. Each test case includes several test actions that are supposed to alter the values of the inputs and compare the

Table 8.6: An Overview of the Comparison between CODESYS Test Manager and CoUnit based on the Validated Features.

Feature	**Test Manager**	**CoUnit**
Cost	MIX *Commercial license, but free to use for academic purposes	FREE *Open Source license
Supported Platforms	Microsoft Windows	Microsoft Windows
Industrial Use	HIGH	LOW
Stage of Development	MATURE *8 versions released so far	PARTIAL *3 versions released so far
Documentation and Report Generation	COMPLETE	SUMMARIZED
Playback Record	YES *Can be realized via the Test Progress feature	NO
Test Suite Support	YES *.tsd, .xml extensions are supported	YES *.xml extension is supported
Test Suite Extension	EASY *New test cases can easily be developed using the available graphical test commands. *One instantiation of the POU under test can be used in all new test suites and test cases *The number of test cases inside a test suite is not limited	HARD *New test cases need to be developed in ST language *For every new test case a new distinct instantiation of the POU under test is required *Every test suite can only contain 100 test cases
Teamwork Support	NO	NO
DevOps/ALM Integration Support	No Information Provided	No Information Provided
Continuous Integration (CI) Support	Yes	Yes
Script Language	Python, All IEC 61131-3 Supported Programming Languages	Structured Text (ST)
Availability of Customer Support:	YES(Official CODESYS customer support is available)	NO
Quality of Documentation	VERY GOOD *Both tool documentation and official tutorial videos are available online	GOOD *Tool documentation and textual tutorial are available online
Maintenance Support	EASY *Direct testing of the PLC main program is supported *GUI and graphical test commands are available	HARD *Direct testing of the main PLC program is not supported *GUI and graphical test commands are not available

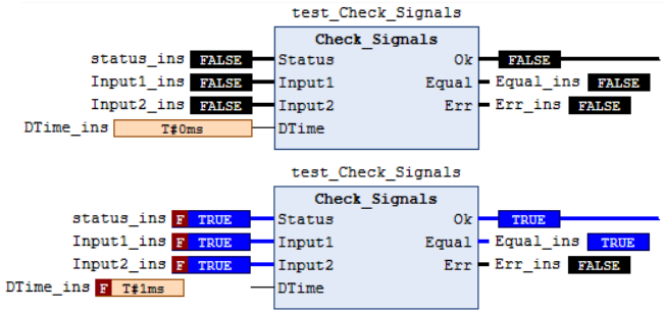

Figure 8.3: An example of the POU under test before (top) and after execution (bottom) of the developed test scripts in CODESYS Test Manager

output of the PLC program with the expected result. For example, five test cases target the functionality of the TON block. In these test cases, we provide just one active input signal with the expected output *Err* being true. The subsequent five test cases use two active signals at different random time slots. After executing the developed test scripts in CODESYS Test Manager, the results of running the PLC program can be observed during execution as shown in Figure 8.3. After running the test suite in CODESYS Test Manager, we observed all the test cases passing, as can be seen in Figure 8.4. The CODESYS Test Manager automatically generates a test report in HTML which includes information on e.g., test settings, test result and status, execution time, and pinned scripts.

To evaluate the applicability of CoUnit, we developed ten different test cases. Since CoUnit does not contain any GUI interface facilitating the creation of test cases, we developed the test scripts in the ST language. Moreover, unlike CODESYS Test Manager, CoUnit does not support the use of the entire Program as a testable object type, and it only supports this functionality at the POU level. We instantiate the POU for every defined test case. In addition, the framework needs to be added as a library into the target PLC program. Also, for each POU under test, the user needs to develop a dedicated function block as a test counterpart, which is responsible for four main tasks, including instantiating the POU under test, defining the inputs, describing the expected output, and finally, calling the CoUnit assertion methods to compare the expected output with the actual one. A snippet of the developed test suite for the CoUnit

Figure 8.4: A generated test report in CODESYS Test Manager

test automation framework implemented in the ST programming language is shown in Figure 8.5. After running CoUnit in the main program (PRG_Test in Figure 8.5), the framework automatically executes the defined test cases on the POUs under test. When the test execution ends, CoUnit generates a test report in the XML format that provides users with information about the test results, including the test suite name, the number of test cases, the test case name, test case status (PASS or FAIL), and the class name.

Finally, we report our overall experiences in using both test automation frameworks. The following results and features are PLC-specific. Regarding **installation and configuration**, we found out that setting up CODESYS Test Manager seems to be more straightforward since it can be installed as a standard add-on package. On the other hand, CoUnit needs to be installed as

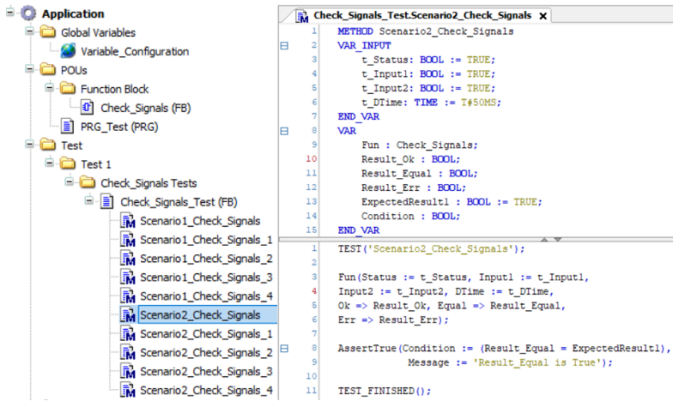

Figure 8.5: A test suite developed in the CoUnit

a package and imported as a library in every project under test. Regarding the **ease of use**, CODESYS Test Manager is more user-friendly and provides features for developing test scenarios using available test commands in the GUI integrated into CODESYS IDE. Moreover, developing test cases with this framework does not require the use of any of the IEC61131-3 programming languages. On the other hand, creating the same test cases in CoUnit is more time-consuming due to the use of ST scripts and instantiations. When comparing the frameworks' capabilities related to **testable objects**, we found out that the Test Manager can create harnesses for PLC applications, IEC libraries, and communications. In contrast, CoUnit can only be used at the application level. Regarding **test assertion timeouts** we note here that PLC programs are executed cyclically in a loop, and one needs to set a test assertion timeout to make sure that the result comparison process ends after a certain amount of time. Only CODESYS Test Manager can be used to set a custom timeout, a useful feature when testing complex PLC programs. After executing test scripts on both frameworks, we discovered that test reports generated by CODESYS Test Manager provide the user with detailed information. On the other hand, CoUnit only reports scarce information.

Results RQ4: Using the discovered features as a basis, the application on an industrial PLC program revealed that both frameworks provide proper

automation functionality. However, CODESYS Test Manager seems to be more mature, provides more helpful test execution features, and is more user-friendly. In contrast, CoUnit seems limited in its usefulness, and working with it requires ST programming.

8.5.5 Threats to Validity

In this section, we discuss some of the threats to the validity for this study. To address *internal validity*, we iterated on the search strings by conducting several initial searches. Aiming to minimize the bias in the process of interpretation, analysis, and selection of the gathered sources, we made sure that at least two authors of this paper reviewed each source. The extraction method we used for filtering and categorizing the gathered data in the pool of contents is based on the systematic qualitative analysis approach proposed by Huberman et al. [12]. When considering the *construct validity* of our study, we employed already proposed methods [5], [4]. Consequently, the data has been examined and checked multiple times to realize an agreement on the obtained features in this study. Regarding *external validity*, since this research is conducted in a very specific domain and it only focused on test automation tools of a particular PLC IDE, more studies are needed to generalize the process of choosing a test automation framework.

8.6 CONCLUSIONS and FUTURE WORK

This paper addresses the practical problem of choosing the right test automation tool for PLC programs in CODESYS IDE. First, we identified the most-discussed test automation tools of CODESYS by performing a GLR on existing test automation frameworks of CODESYS followed by a qualitative analysis based on several criteria. Aiming at performing an effective comparison between the discovered test automation frameworks of CODESYS, we identified 29 features as important features of test automation tools by conducting a literature review. Finally, to investigate the applicability of the discovered tools in a real-world case study, we performed an automated test execution on an industrial case study based on two different test scenarios. Our findings imply that both discovered test automation tools of CODESYS provide users with nec-

essary automation functionalities but CODESYS Test Manager seems more mature, has more useful test execution features, and is more user-friendly. By contrast, CoUnit is not user-friendly, is limited in its automation features, and working with it demands ST programming knowledge. In future research, we plan to conduct a more comprehensive evaluation of CODESYS Test Manager when used in an industrial context as well as how to connect such a test automation framework for automated testing of timers and stateful blocks in PLCs.

8.7 Acknowledgment

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

Bibliography

[1] Irfan Ahmed, Sebastian Obermeier, Sneha Sudhakaran, and Vassil Roussev. Programmable logic controller forensics. *IEEE Security & Privacy*, 15(6):18–24, 2017.

[2] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[3] Eliane Figueiredo Collins and Vicente Ferreira de Lucena. Software test automation practices in agile development environment: An industry experience report. In *International Workshop on Automation of Software Test (AST)*, pages 57–63. IEEE, 2012.

[4] Vahid Garousi, Wasif Afzal, Adem Çağlar, İhsan Berk Işık, Berker Baydan, Seçkin Çaylak, Ahmet Zeki Boyraz, Burak Yolaçan, and Kadir Herkiloğlu. Comparing automated visual gui testing tools: an industrial case study. In *International Workshop on Automated Software Testing*, pages 21–28, 2017.

[5] Päivi Raulamo-Jurvanen, Mika Mäntylä, and Vahid Garousi. Choosing the right test automation tool: a grey literature review of practitioner sources. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 21–30, 2017.

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

[7] Dag H Hanssen. *Programmable logic controllers: a practical approach to IEC 61131-3 using CODESYS*. John Wiley & Sons, 2015.

[8] Marcin Jamro. Pou-oriented unit testing of iec 61131-3 control software. *IEEE Transactions on Industrial Informatics*, 11(5):1119–1129, 2015.

[9] Florian Hofer and Barbara Russo. Iec 61131-3 software testing: A portable solution for native applications. *IEEE Transactions on Industrial Informatics*, 16(6):3942–3951, 2019.

[10] Päivi Raulamo-Jurvanen, Simo Hosio, and Mika V Mäntylä. Practitioner evaluations on software testing tools. In *Proceedings of the Evaluation and Assessment on Software Engineering*, pages 57–66. 2019.

[11] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 1–6, 2016.

[12] A Michael Huberman and Johnny Saldana Matthew B Miles. *Qualitative data analysis: A methods sourcebook*. 2019.

[13] Alessio Ferrari, Franco Mazzanti, Davide Basile, and Maurice Ter Beek. Systematic evaluation and usability analysis of formal methods tools for railway signaling system design. *IEEE Transactions on Software Engineering*, 2021.

[14] Mubarak Albarka Umar and Chen Zhanfang. A study of automated software testing: Automation tools and frameworks. *International Journal of Computer Science Engineering (IJCSE)*, 6:217–225, 2019.

[15] Heidilyn Veloso Gamido and Marlon Viray Gamido. Comparative review of the features of automated software testing tools. *International Journal of Electrical and Computer Engineering*, 9(5):4473, 2019.

[16] Harpreet Kaur and Gagan Gupta. Comparative study of automated testing tools: selenium, quick test professional and testcomplete. *Journal of Engineering Research and Applications*, 3(5):1739–1743, 2013.

[17] Emil Borjesson and Robert Feldt. Automated system testing using visual gui testing tools: A comparative study in industry. In *International Conference on Software Testing, Verification and Validation*, pages 350–359. IEEE, 2012.

[18] Sebastian Ulewicz and Birgit Vogel-Heuser. Increasing system test coverage in production automation systems. *Control Engineering Practice*, 73:171–185, 2018.

[19] Sebastian Ulewicz and Birgit Vogel-Heuser. Guided semi-automatic system testing in factory automation. In *International Conference on Industrial Informatics (INDIN)*, pages 142–147. IEEE, 2016.

[20] Sebastian Ulewicz and Birgit Vogel-Heuser. System regression test prioritization in factory automation: Relating functional system tests to the tested code using field data. In *Annual Conference of the IEEE Industrial Electronics Society*, pages 4619–4626. IEEE, 2016.

[21] Sebastian Ulewicz and Birgit Vogel-Heuser. Industrially applicable system regression test prioritization in production automation. *Transactions on Automation Science and Engineering*, 15(4):1839–1851, 2018.

[22] GIACOMO Barbieri, GABRIEL Quintero, OSCAR Cerrato, JULIAN Otero, DAVID Zanger, and ALEJANDRO Mejia. A mathematical model to enable the virtual commissioning simulation of wick soilless cultivations. *J. Eng. Sci. Technol*, 16:3325–3342, 2021.

Chapter 9

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

Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu
Published in The 38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023).

9.1 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.

9.2 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 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) [1].

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 [2]. 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 9.3 briefly overviews PLC programming languages FBD and ST, CODESYS, Python, as well as Pynguin. Section 9.4 describes the translation process, translation challenges, and translation validation mechanisms of the PyLC framework. Section 9.5 explains the gathered results of this study regarding each formulated research question as well as threats to validity. Section 9.6 overviews the related work. Finally, section 9.7 briefly overviews the conclusions and potential future research directions.

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

9.3 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.

9.3.1 PLC Programming, IEC 61131-3, and CODESYS

IEC 61131-3 standard [1] 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) [1]. 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.

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 [1]. An FBD representation consists of three main parts, including (i) the *POU*, (ii) a declaration, and (iii) the actual code representing the behaviour [1]. 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 bit-

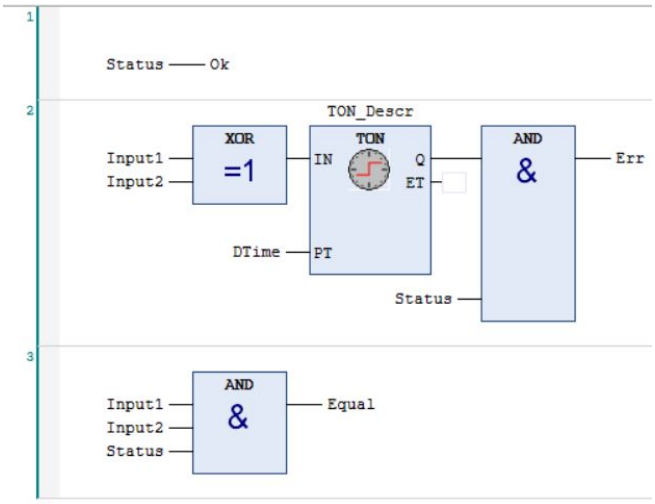

Figure 9.1: A Snippet of an example PLC Program (*Check_Signals*) written in FBD.

wise boolean functions), selection and comparison functions, character string functions, functions for time data types, and functions for enumerated data types [1]. 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 9.1. 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

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

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.

Structured Text (ST)

ST is one of the most popular text-based programming languages of the IEC 61131-3 standard [1]. 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 9.2 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 9.2, the program consists of variables/data type declaration using the assignment statements, while the

program logic concerning the execution order is written separately.

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.

9.3.2 Python and Pynguin

Python

Python is an open-source dynamic programming language that was invented by Guido van Rossum in 1990. The motivation behind creating this programming language was to produce an advanced scripting language for the Amoeba system [3]. 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 the 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.

Pynguin Test Automation Framework

Pynguin [2] 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 [4], DYNAMOSA [5], MIO [6], and WHOLE SUITE [7]. It is also equipped with a random test generator named RANDOM [8], which works based on the RANDOOP algorithm [9]. We note here that Python is the only non-IEC 61131-3 programming language officially supported by CODESYS IDE (a well-known

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

PLC IDE) and can be directly compiled inside the IDE. For more information about Pynguin, we refer the reader to the following publications [2], [10], [11].

9.4 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 9.3 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 9.3). 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 9.3). Finally, the Translation Validation module of the transformed code (Step 4 in Figure 9.3) uses unit testing to ensure that the code is scrutinized for proper use in further analysis and test generation.

9.4.1 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 9.4. 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 9.4). 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 9.4). In the next step,

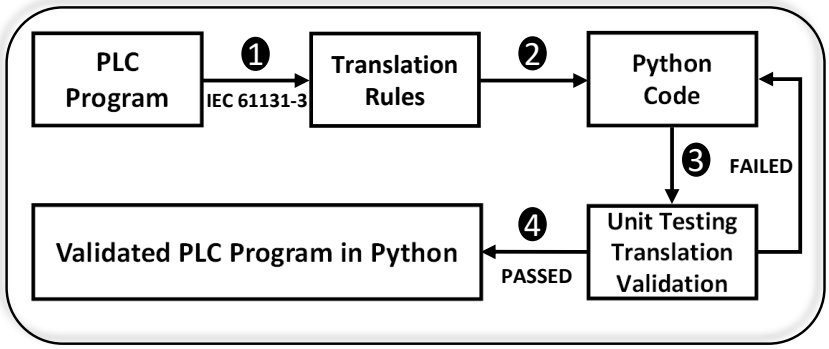

Figure 9.3: An Overview of the PyLC Framework, the Proposed Translation Mechanism for Translating a PLC Program into Python Code and Validating the Translation.

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 9.4). 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 9.4). The final step is identifying the execution order of the program elements inside the PLC program and implementing it in the translated Python code (Steps F, G in Figure 9.4).

An overview of the translation rules we adhere to in the translation process is observed in Table 9.1. 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.

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*),

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 behaviour of the FB based on the requirements	Implementing the FB behaviour 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 9.1: Translation Rules (TR) of the Proposed PLC Program to Python Code Considering IEC-61131-3 Standard

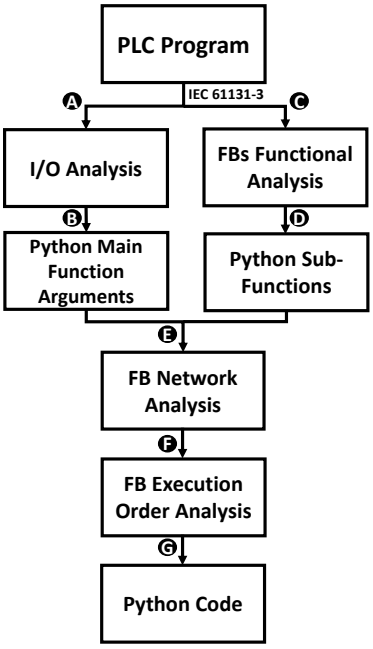

Figure 9.4: The Translation Work Flow (TWF) Used in PyLC Framework for Translating a PLC Program into Python.

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 9.5).

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.

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.

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 [1].

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 *Safe-Supervision* 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 9.1. The step-by-step translation process of this example is shown in Table 9.2. Based on the translation methodology described in Figure 9.4, 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 9.1, 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 *Safe-Supervision*. This program is described in Section 9.3. Based on the proposed translation mechanism in Figure 9.4, the first step is to detect the inputs and outputs of the program as well as their data type. In this program, the inputs are

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

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 9.5.

9.4.2 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 consists of 8 main steps and can be observed in Figure 9.6. 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 generated manually for a translated PLC program into Python (Step 1 in Figure 9.6). 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 9.6). 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 9.6).

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 9.6). 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

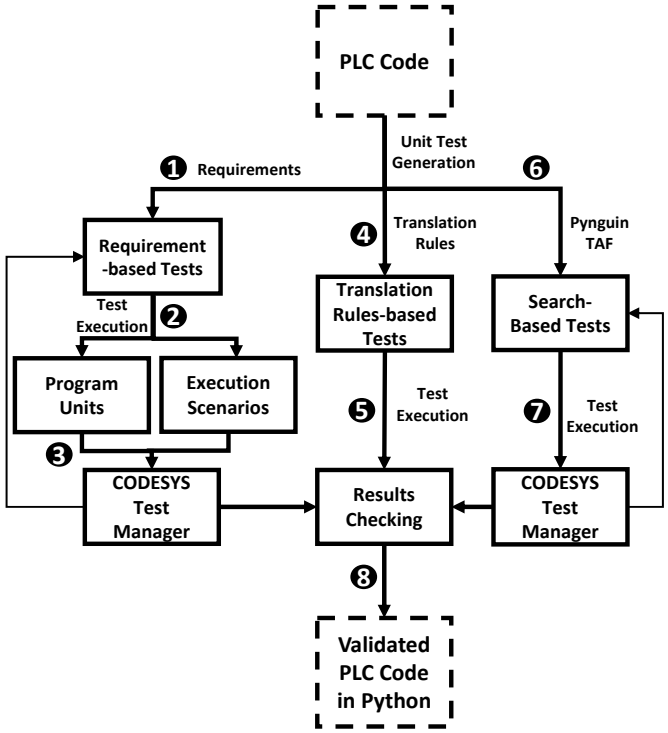

Figure 9.6: An Overview of the Hybrid Unit-Testing Validation Mechanism of the Translated PLC Code in Python

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

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

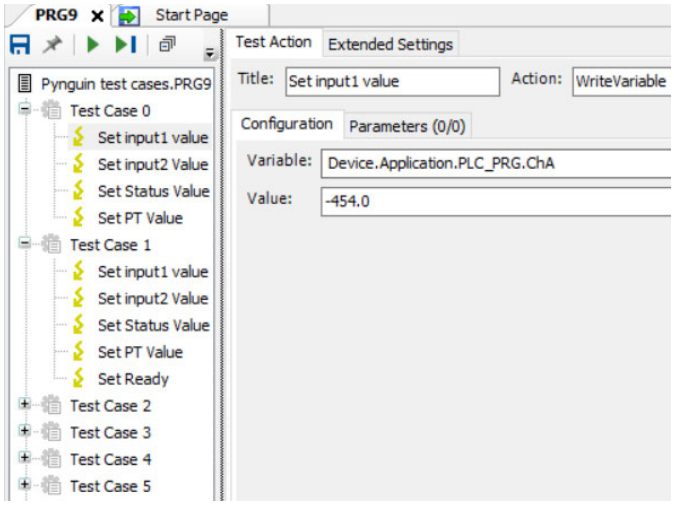

Figure 9.7: A Snippet of the Written Test Cases in CODESYS Test Manager for a PLC program.

the main PLC program to be used by the CODESYS Test Manager. Finally, the PLC device login is completed, and CODESYS Test Manager test scripts are executed on the original PLC program. A snippet of the implemented test cases in CODESYS Test Manager can be seen in Figure 9.7.

Validation by Translation Rules Checking: Evaluating the proposed translation rules in Table 9.1 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 9.6). 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 9.6). 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 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 9.6) that is equipped with different search-based algorithms [2]. 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 9.6). 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 9.6), the proposed code validation process is completed, and the resulting translation results are validated (Step 8 in Figure 9.6).

Table 9.2: 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).

<table>
<tr><th>TWF Step</th><th>Step Name</th><th>Related Translation Rule</th><th>PLC Code Description</th><th>Translated Code in Python/Description</th></tr>
<tr><td>A/B</td><td>I/O Analysis</td><td>Inputs
Outputs
Data Type
Data Range</td><td>Input1(bool), Input2 (bool)
Q(bool), Status (bool)
IN(bool), PT(TIME), Q(bool)</td><td><pre>#Python Main Function
def Check_Signals(input1: bool, input2: bool,
          status: bool, pt: int) -> bool:
    if pt < 0:
        raise ValueError</pre></td></tr>
<tr><td rowspan="4">C/D</td><td rowspan="4">FBs Functionia l Analysis</td><td rowspan="4">FB Behavior</td><td>XOR (Input1, Input2)</td><td><pre>#Python Sub-Function (Dynamic range of Arguments)
def XOR(*args):
    val = args[0]
    for i in range(1, len(args)):
        if type(args[i]) is not bool:
            raise TypeError
        else:
            val = val ^ args[i]
    return val</pre></td></tr>
<tr><td>AND(Q, Status)</td><td><pre>#Python Sub-Function (Dynamic range of Arguments)
def AND(*args):
    val = args[0]
    for i in range(1, len(args)):
        if type(args[i]) is not bool:
            raise TypeError
        else:
            val = val and args[i]
    return val</pre></td></tr>
<tr><td>TON(IN, PT, Q)</td><td><pre>#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</pre></td></tr>
<tr><td>AND(Input1, Input2, Status)</td><td><pre>#Python Sub-Function (Dynamic range of Arguments)
def AND(*args):
    val = args[0]
    for i in range(1, len(args)):
        if type(args[i]) is not bool:
            raise TypeError
        else:
            val = val and args[i]
    return val</pre></td></tr>
<tr><td>E</td><td>FB Network Analysis</td><td>FB Network</td><td>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.</td><td><pre>#The connection between the Python code elements including
  FBs, inputs, and outputs established using the Python
              function calls in right order</pre></td></tr>
<tr><td rowspan="2">F</td><td rowspan="2">FB Execution Order Analysis</td><td rowspan="2">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)</td><td>Excecution 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</td><td rowspan="2"><pre># 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</pre></td></tr>
<tr><td>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.</td></tr>
<tr><td>G</td><td>Python Code</td><td>Cyclic Execution</td><td>Finally, INPUTS, OUTPUTS, NETWORK, and EXECUTION ORDER are linked to each other to generate the trasnlated Python Code.</td><td><pre>#Cyclic execution is implemented by calling a Python timer
          module with a preset time budget.</pre></td></tr>
</table>

9.5 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.

9.5.1 RQ1 - PyLC Translation

We consider ten different PLC programs to evaluate our proposed translation framework in real-world circumstances, including 6 ST and 4 FBD programs. Detailed information on the translated PLC programs is shown in Table 9.3. 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 9.4 and adheres to the proposed translation rules in Section 9.4.1. We note here that, according to the data in Table 9.3, 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.

9.5.2 RQ2 - PyLC Validation

To evaluate the proposed method, we use the translation results of the translated PLC programs in Section 9.5.1 by three different unit testing mechanisms described in Section 9.4.2. In the following subsections, we describe and demonstrate the results regarding each unit testing validation step, respectively.

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 9.3: 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 9.4: Results of executing the test cases for each common Program (PRG) unit as well as their type: Function (FUN)/Function Block(FB)

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

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.

Based on the proposed technique for this type of validation (as shown in Figure 9.6), we analyze the behaviour of the translated code from two different aspects, which 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 9.5 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 9.4 shows more details and results of testing these blocks. As it can be observed in Table 9.4, 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 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.

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

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

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 9.6.

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 minutes, 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 9.7.

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

As seen in Table 9.7, 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 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-minute 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

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

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.

9.5.3 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 these 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].

9.6 Related Work

Previous contributions in transforming PLC programs to other languages range from SCs-based approaches (e.g., [12]) and the ones using the C language (e.g., [13]) to model-based approaches of transforming the actual FBD program code

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

(e.g., [14]). The technique in [15] is based on the IEC 6150 models and supports other parts of the development process. However, compared to our work, these works do not cope 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. [12] 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 [16]. 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. [14] proposed a toolbox that can formalize logic coverage criteria and use it inside a model-checker to generate test cases [14]. 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. [13] 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 data-flow 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 [15], Mirko Conrad [17] 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.

9.7 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 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.

9.8 Acknowledgements

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

Bibliography

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

[2] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In *International Symposium on Search Based Software Engineering*, pages 9–24. Springer, 2020.

[3] Mark Lutz. *Programming python*. " O'Reilly Media, Inc.", 2001.

[4] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[5] 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 Engineering*, 44(2):122–158, 2017.

[6] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. In *International symposium on search based software engineering*, pages 3–17. Springer, 2017.

[7] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[8] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007.

[9] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[10] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for python. *arXiv preprint arXiv:2111.05003*, 2021.

[11] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. *arXiv preprint arXiv:2202.05218*, 2022.

[12] Marcel Christian Werner and Klaus Schneider. From iec 61131-3 function block diagrams to sequentially constructive statecharts.

[13] Junbeom Yoo, Eui-Sub Kim, and Jang-Soo Lee. A behavior-preserving translation from fbd design to c implementation for reactor protection system software. *Nuclear Engineering and Technology*, 45(4):489–504, 2013.

[14] 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(3):335–353, 2016.

[15] Ron Bell. Introduction to iec 61508. In *Acm international conference proceeding series*, volume 162, pages 3–12. Citeseer, 2006.

[16] Klaus Schneider. The synchronous programming language quartz. Technical report, Internal Report 375, Department of Computer Science, University of Kaiserslautern, 2009.

[17] Mirko Conrad. Testing-based translation validation of generated code in the context of iec 61508. *Formal Methods in System Design*, 35(3):389–401, 2009.

Chapter 10

Paper C: An Empirical Investigation of Requirements Engineering and Testing Utilizing EARS Notation in PLC Programs

Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, Cristina Seceleanu
Submitted to the Springer Nature Journal's Special Issue on Topical Issue on Advances in Combinatorial and Model-based Testing 2023.

10.1 Abstract

Regulatory standards for engineering safety-critical systems often demand both traceable requirements and specification-based testing, during development. Requirements are often written in natural language, yet for specification purposes, this may be supplemented by formal or semi-formal descriptions, to increase clarity. However, the choice of notation of the latter is often constrained by the training, skills, and preferences of the designers.

The Easy Approach to Requirements Syntax (EARS) addresses the inherent imprecision of natural language requirements with respect to potential ambiguity and lack of accuracy. This paper investigates requirements specification using EARS, and specification-based testing of embedded software written in the IEC 61131-3 language, a programming standard used for developing Programmable Logic Controllers (PLC). Further, we study, by means of an experiment, how human participants translate natural language requirements into EARS and how they use the latter to test PLC software. We report our observations during the experiments, including the type of EARS patterns participants use to structure natural language requirements and challenges during the specification phase, as well as present the results of testing based on EARS-formalized requirements in real-world industrial settings.

10.2 Introduction

Programmable Logic Controllers (PLCs) are used in engineering embedded safety-critical software (e.g., in the railway and automation control domains) [1]. Engineering such systems commonly demands certification according to safety standards [2] that impose specific constraints on requirements engineering, implementation-based and specification-based testing. Several studies [3], [4], [5], [6] have looked at how to generate test input data to achieve high implementation coverage for domain-specific PLC systems.

However, since requirements are often expressed in natural language, using them as such to create test cases, and also keep requirements and test cases aligned, is a difficult task. While such an alignment requires extensive domain knowledge, a systematic process for requirements engineering – including their translation into a semi-formal, non-ambiguous form – combined with testing would facilitate linking requirements to tests. Generally, in industry,

such translation is most often carried out manually, so manual processes are used to model requirements by using structured notations, and automatically create a set of tests that systematically exercises the specification when fed to the system under test [7]. Given that there is little evidence on the extent to which humans can effectively model requirements using semi-formal notations, and how the modelling impacts the development and testing of reliable systems, in this paper, we investigate the implications of applying structured requirements specification and test generation based on the latter, for PLC systems. In this context, we study how practitioners write requirements using the *Easy Approach to Requirements Syntax* (EARS) [8], a simple notation for specifying textual requirements in a structured and unambiguous manner.

We evaluate the EARS-based requirement modelling by involving human subjects. Ten individuals take part as subjects, in an experiment. The subjects are given three requirements specified in natural language, and are asked to rewrite them manually, using the EARS notation.

This work builds upon our previous work [9], and it extends it by a rigorous investigation of the applicability and efficiency of EARS-based testing of industrial PLC programs, including a comparison of EARS-based testing of PLC programs with testing the same programs manually. The results of our study show that humans create pattern-based requirements using semi-formal notations easily, with completeness being the most common issue when rewriting and using such requirements for testing. Additionally, we find that test generation and execution using the EARS requirements for PLC systems is a promising approach that is applicable to real-world industrial settings. Our results highlight the need for more research into how different requirement specifications and test design techniques for PLC software can influence the efficiency and effectiveness of requirements engineering and requirements-based testing for this type of software.

10.3 PRELIMINARIES

10.3.1 Programmable Logic Controllers

Programmable Logic Controllers (PLC) are the most used logic controllers in today's automation industry [10]. PLCs are being widely used in different industrial applications such as supervisory systems in nuclear and power plants.

Programming a PLC device is usually done via one or a combination of different programming languages that are proposed in the IEC 61131-3 standard [11], that is, *Function Block Diagram* (FBD), *Structured Text* (ST), *Ladder Diagram* (LD), *Sequential Function Chart* (SFC), and *Continuous Function Chart* (CFC). Among all programming languages for PLC, FBD and ST are our main focus in this study, for two reasons. First, these two languages have gained remarkable popularity in industry, during the last couple of years [12]. Second, the industrial case study that is provided to us for this study is a supervisory PLC program developed in ST and FBD. ST is a text-based programming language with a similar syntax to high-level programming languages such as C, whereas, FBD is a visual programming language that is easy to use due to its graphical interface. PLC programs are commonly developed in an Integrated Development Environment (IDE) and are executed cyclically. Based on the provided concept in IEC 61131-3, each cycle loop of a PLC program execution consists of 3 main stages, that is, read, execute, write [11]. The first stage reads all available inputs and stores them in the memory, whereas the second stage (execute) carries out the computation tasks without interruption. The final stage (write) updates the output values based on the completed computations of the previous stage.

10.3.2 CODESYS Development Environment

Developing a PLC program and simulating its behaviour needs to be done in an IDE. Several different PLC IDEs have been proposed by different vendors so far. One of the most popular IDEs in the market is CODESYS [1] which was initially developed by CODESYS Group in 1994. CODESYS is a manufacturer-independent IDE that has matured by releasing numerous updates and the latest version at this moment is V3.5 SP18. Among all available PLC IDEs in the market, We have chosen CODESYS as our preferred IDE because of several reasons. Firstly, CODESYS is very popular among practitioners and has almost full compatibility with the IEC61131-3 standard and supports all proposed standard programming languages of this standard [12]. Secondly, CODESYS is free to use for personal use and is equipped with good support through releasing different versions. Last but not least, CODESYS can execute Python scripts directly inside the IDE and it is also equipped with numerous

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

automation add-ons such as test automation tools.

10.3.3 EARS Semi-Structured Requirement Engineering Syntax

Writing the stakeholder requirements in unconstrained Natural Language (NL) is not accurate and can raise critical problems in lower levels of system development [8]. Aiming at mitigating the ambiguity problems and increasing the accuracy in the process of requirements engineering, some practitioners stand up for using other textual and non-textual notations [8]. Using non-textual notations demands translation of the original requirement, which can be faulty sometimes. Training overhead is another drawback of proposing a new type of notation. EARS is a semi-structured requirement engineering syntax that was proposed by Alistair et al. in 2009 [8]. EARS provides a syntax for transforming all-natural language requirements in one of the proposed five Generic requirements syntax simple templates. The aforementioned five simple templates of EARS are ubiquitous requirements, event-driven requirements, unwanted behaviours, state-driven requirements, and optional features. Moreover, EARS supports writing complex requirements using a combination of considered conditional keywords, including *Where*, *While*, and *When*.

10.4 EXPERIMENTAL DESIGN

In this section, we report the description of the performed experiment, including the details of the instruction material and the artefacts used.

10.4.1 Research Questions

The main goal of this study is to investigate the process of requirements creation when constraining the use of NL. The EARS modelling notation has been adopted by other organizations in different sectors and countries, so it is a realistic model for requirements engineering and test creation. Since these are intellectual activities in which humans allocate a variety of cognitive resources (such as attention and effort) that one needs to use when confronted with challenges as they perform such tasks, our first step is understanding how human

practitioners write such requirements and how these can be used for test creation.

The main goal of this study is to investigate the applicability of the EARS semi-structured requirement engineering syntax in the context of PLC programs. Aiming at achieving this goal, we formulated the following research questions.

- RQ1: How is the EARS semi-structured requirement engineering syntax and test creation applied in the context of PLC programs?
- RQ2: What EARS patterns are used during the writing of requirements?
- RQ3: What challenges are perceived during the specification of requirements and test creation using EARS?
- RQ4: How well do PLC test cases created from EARS requirements compare to test cases created by industrial engineers for PLC programs in industry?

10.4.2 Experimental Setup Overview

Aiming at achieving the goal of this study, we conduct a controlled experiment that asks the participants to write 3 given requirements using EARS syntax. The participants are free to choose their preferred EARS syntax template based on their personal interpretation of the given requirements. The *subjects* of this experiment are a group of 10 individuals as follows: four experienced engineers at a large automation company in Sweden and Spain and six researchers and managers from different universities and research institutions across Europe.

10.4.3 Object Selection

The objects of study were chosen manually based on the following criteria:

- The requirements should have a natural language specification that is understandable and sufficiently rich in detail for an engineer to write executable tests.
- The requirements should represent different types of real testing scenarios in different areas where the IEC 61131-3 standard is used.

Table 10.1: The natural language requirements used during the experiment.

Requirement ID	Requirement Text
RI1	User account should be uniquely identified to a user.
RI2	The software shall warn the user of malware detection.
RI3	Only authorised devices are allowed to connect into the ICS network

- The requirements should be simple to understand without any domain knowledge.
- The resulting test cases should be executed in the CODESYS environment.

We investigated the industrial libraries provided by a large-scale company focusing on the development and manufacturing of control systems. We identified three candidate requirements matching our criteria, shown in Table 10.1. The requirements should not be trivial, yet fully manageable to use within 60 minutes and no domain-specific knowledge should be needed to understand the requirements. We then assessed the relative difficulty of the identified requirements by manually writing and creating tests.

10.4.4 Operationalization of Constructs

Requirements Templates. In this experiment, we investigate the effect of using the EARS approach for requirements engineering and test creation. The proposed generic requirements syntax of EARS we used in this experiment works as follows:

<optional preconditions><optional trigger> the <system name> shall <system response>

This simple syntax template forces the requirement engineer to emphasise preconditions, triggers, and system responses in their developed requirements. In EARS syntax, preconditions, and triggers are both optional, and the order

of the used clauses is very important. The following briefly describes each template of EARS.

Ubiquitous requirements (U)

A ubiquitous requirement is a type of requirement that is not bonded to any preconditions or triggers and is always enabled in the system. The generic structure of this template is as follows:

The <system name> shall <system response>

Event-driven requirements (ED)

The event-driven requirement is used only when an event is identified in the system. This type of requirement uses *When* keyword. The generic structure of this template is as follows:

WHEN <optional preconditions> <trigger> the <system name> shall <system response>

Unwanted behaviours (UB)

Requirements that are related to Unwanted behaviours are defined using a structure that is extracted from Event-driven requirements. *Unwanted behaviour* refers to covering all possible situations that are not desirable and are usually a big source of omissions in preliminary requirements. The reserved keywords for this type of requirement in EARS are *If* and *Then*. The generic structure of this template is as follows:

IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>
State-driven requirements (SD)

The State-driven requirement is only active if the system is in a specific status. The reserved keyword for defining State-driven requirements in EARS is *While*. The generic structure of this template is as follows:

WHERE <feature is included> the <system name> shall <system response>

Optional features (OF)

The *Optional feature* requirement is designed to be used when the author of the requirement wants to include a specific feature in the system. The keyword *Where* is considered for defining this type of feature in EARS. The generic structure of this template is as follows:

WHERE <feature is included> the <system name> shall <system response>

Process Challenges. We are interested in two types of challenges encountered during the use of EARS templates and their use for testing: challenges encountered during the specification of requirements and problems when designing test cases for PLC systems. We performed thematic analysis [13] for qualitative data analysis to extract the main themes as reflected by the input given by each participant.

10.4.5 Instrumentation

One session was organized for the sake of the experiment. The subjects were given the task to use the three requirements and rewrite these in EARS (to the extent they consider sufficient based on the given specifications). They were instructed to read the specifications, create these templates and think out loud. The subjects were not grouped and the document needed for this experiment was provided digitally and in written form. Before commencing the session, a short tutorial of approximately 10 minutes on EARS syntax was provided to the subjects in order to avoid further problems with the subjects' unfamiliarity with the concepts used. The tutorial included screencasts demonstrating EARS requirements. Detailed information about the problem and instructions were provided in the experiment session.

10.4.6 Data Collection Procedure

As part of the instructions, subjects submitted their solutions in the form of a record documenting their work. Data from this experiment session was then

used for quantitative and qualitative analysis.

10.5 EXPERIMENT CONDUCT

Once the experiment design was defined, the requirements for executing the experiment were in place. The session was held for one hour and preceded by a lesson on EARS notation. The requirements specification and testing process used during the conduct of this experiment corresponds to the methodology in Figure 10.1. The first step corresponds to the transformation of the requirement specified initially in Natural Language (NL) into an EARS requirement using the EARS syntax (Step 1 in Figure 10.1). In the next step, we are using the resulting requirement to generate test cases that cover the specified behaviour (Step 2 in Figure 10.1). The final steps in this methodology are to execute these test cases (Step 3 in Figure 10.1) and to compare the actual behaviour with the expected result to monitor whether the program works as expected (Step 4 in Figure 10.1).

In total, *ten individuals* participated in our experiment. Before starting the experiment, the participants were informed that their work would be used for experimental purposes. The participants had the option of not participating in the experiment and not allowing their data to be used this way.

The subjects worked individually during the experiment; we briefly interacted with the participants to ensure that everybody had a sufficient understanding of the involved notations without getting involved in the writing of the solution. All subjects used the provided documents and their machines. The experiment was fixed to one hour. To complete the assignment, the subjects were given the same time to work on writing these requirements according to the given instructions. For collecting data, we provided a template to enforce the usage of the same reporting interface. By having a common template for reporting, we eased the data collection and analysis process.

To finish the assignment, we required the participants to provide the produced results as soon as they finished writing their responses. During the experiment, the subjects do not directly communicate with others to avoid introducing bias. After each individual finished their assignment, a complete solution was saved containing the answers for each solution. In addition, we separated the data provided by the participants from their names.

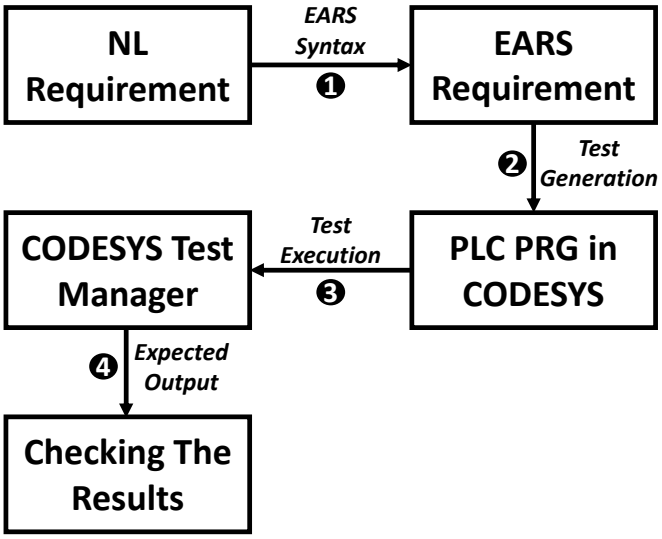

Figure 10.1: An overview of the proposed EARS-based requirement specification and PLC testing methodology used in this experiment.

10.6 EXPERIMENT ANALYSIS

This section provides an analysis of the data collected in this experiment. In analyzing the qualitative data, we followed the guidelines on qualitative analysis procedures provided by Braun and Clarke [13]. For each requirement, each subject in our study provided a set of EARS expressions. These expressions were used to conduct the experimental analysis and testing. For each set of tests produced, we provide evidence for their generation and execution in CODESYS. These metrics form the basis for our analysis toward answering the research questions.

10.6.1 Requirement Engineering Results

For each requirement, we have collected data about the type of EARS template used by each participant, the approaches, and the challenges participants experienced during requirement representation using the EARS notation. The results are shown in Table 10.2, Table 10.3, and Table 10.4.

Table 10.2: Results of the templates used for each requirement used in the experiment.

RI1	RI2	RI3	Requirement ID/EARS Template
10	1	1	**Ubiquitous (U)**
0	5	4	**Event-Driven (ED)**
1	5	6	**Unwanted Behaviours (UB)**
0	0	3	**State-Driven (SD)**
0	0	0	**Optional Features (OF)**

Table 10.3: Results of the requirements writing in terms of the templates used by each participant for each requirement. EARS template types are shown using their specific acronyms as stated in Section 10.4.4 and Table 10.2.

RI1	RI2	RI3	Requirement ID/Participants
U, UB	U, UB, ED	U, SD, ED	**P1**
U	ED	UB	**P2**
U	ED	UB	**P3**
U	UB	SD	**P4**
U	ED	UB	**P5**
U	ED	UB	**P6**
U	SD	UB	**P7**
U	UB	ED, UB, SD	**P8**
U	UB	ED	**P9**
U	UB	ED	**P10**

Table 10.4: Results showing the main themes identified related to approaches and challenges encountered during the translation process.

Main Themes	**Theme Descriptions**
Requirements are not complete and clear enough for EARS translation.	When starting with the translation, requirements in NL are not complete enough to decide precisely which EARS template to use.
Using single or multiple EARS templates is not clear enough, especially when using these for testing.	There is a need, when using these patterns for testing, to use multiple and separate templates for each requirement to cover both positive and negative cases arising.
The system perspective is not easily identifiable from the requirements.	It is difficult to decide which perspective to use when translating the EARS requirement (e.g., system, subsystem level).
The optional feature template is not applicable for the selected requirements	Even if the Option requirement is used for systems that include a particular element and variants, this modelling form was not used during requirement transformation using the EARS notation since the participants did not need to handle system or product variation.

Participants strictly adhered to one or multiple EARS templates. It seems that the ubiquitous template has been used by all participants to model requirement RI1 and just in one case when representing requirements RI2 and RI3 (as shown in Table 10.2). Participants explained that the "shall" statement is clearly indicated and should be used to describe the required behaviour. Nevertheless, one participant decided to use the unwanted behaviour template for RI1 to indicate the prohibited behaviour in such a form that can be used for testing.

The event-driven and unwanted behaviour templates have been used by participants to represent requirement RI2, while some participants used the state-driven pattern (as shown in Table 10.3). Participants chose to do this since they drafted requirements in several increments. Firstly, they considered how the system behaves typically (also called sunny-day behaviour). For some participants using EARS, this results in requirements in the state-driven and event-driven patterns. Secondly, some participants decided to specify what the system must do in response to the unwanted behaviour, which produced requirements in the unwanted behaviour pattern.

In addition, the thematic analysis of the notes taken by participants when performing these steps in requirement representation resulted in several main themes related to approaches and challenges experienced during the translation process. Several participants mentioned that the initial NL requirements are not complete and clear such that these can be used directly for testing. One participant mentioned the following: *"What happens if the device is not authorized, missing failure models, startup/default/safe state...?"*. This resulted in issues when starting with the translation process, especially when deciding which templates to use. Several participants had issues in deciding when to use single or multiple EARS templates to cover both positive and negative behaviours that need to be tested. One participant stated the following: *"We could possibly use event-driven type requirement. At the same time, it is unwanted we could use, this one is quite complicated"*. Some participants preferred the use of the "shall not" form, which has been observed by some participants as having an impact on the test case created since only a set of test cases involving the unwanted behaviour would need to be created to show satisfaction with the requirement. Another observation relates to the use of an optional feature template, which for the given requirements was not used by any of the participants since there was no need to specify any product variation or specific features.

10.6.2 PLC Testing Results

Aiming at evaluating the applicability of using EARS semi-structured syntax when creating test cases for PLC programs, we used three programs that implement the behaviour stated in the three provided natural language requirements used in this experiment. All these three PLC programs are developed in CODESYS IDE using the Structure Text (ST) programming language. In this paper, we refer to these programs as *PRG1*, *PRG2*, and *PRG3*.

After generating the EARS-based test cases for each program, we execute these automatically using the CODESYS test automation framework named CODESYS Test Manager[2]. The final step in this methodology is to compare the actual output with the expected output to observe whether the program works as expected.

We used the concretization steps of the EARS expressions as stated by Flemstrom et al. [14]. This happens by mapping the system response, condition, and events to the actual implementation in PLC. This contains information about the implementation elements of a system and its interfaces. An engineer needs to consider this information and identify the given signals and their characteristics. In this way, we define a set of signals related to the feature under test. In these cases, the next step for the selected requirements would be to design test cases to show that the requirement has been met. In our experiment, we could directly use a subset of positive and negative cases by randomly choosing values from an equivalence class. Nevertheless, in a general case, the translation and concretization steps are not easy and one would need to decide how to automate such steps and if we are to use exhaustive testing, equivalence class testing, combinatorial testing, or any other test selection technique for designing test cases.

Test Results of *PRG1*

PRG1 is the PLC program we considered for testing the RI1 requirement (refer to Table 10.1) in the PLC environment. This program is using the values of the *user account* and *user* lists. Then it checks for unique IDs and returns an indication of whether each user account is uniquely identified to a user or not. A snippet of the PRG1 PLC program is shown in Figure 10.2.

[2]https://store.codesys.com/en/codesys-test-manager.html

```
1  PROGRAM UniqueUserAccount
2  VAR
3     user : ARRAY[1..10] OF WSTRING;;
4     user_account : ARRAY[1..10] OF DINT;
5     i,j : INT;
6     K : INT;
7     UniqueID : BOOL;(*Non-Unique ID counter*)
8     Result_Unique: BOOL := FALSE;
9  END_VAR
```

Figure 10.2: PRG1 PLC interface program written in the ST language in CODESYS IDE corresponding to the evaluation of the RI1 requirement.

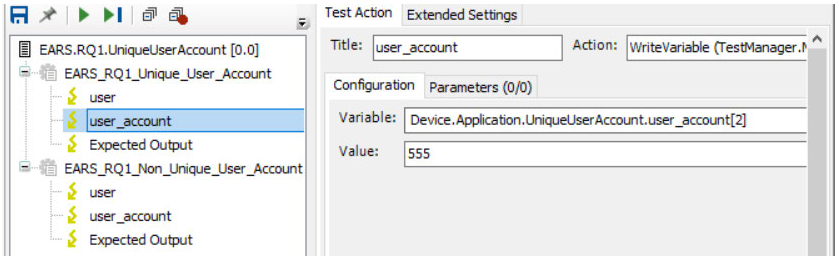

Figure 10.3: The generated test cases for PRG1 based on the EARS syntax for RI1 as shown in CODESYS IDE

To design and execute the required test cases to test the RI1 Requirement in PRG1, we use the transformed requirement from the NL requirement shown in Table 10.5.

Based on the EARS requirement we use two test cases to cover the identification of the user and the case when the user is not identified. Each test case includes the following three test actions: two *WriteVariable* test actions to alter the *user* and *user account* inputs and one *CompareVariable* test action that compares the actual output with the expected one. The generated test cases for PRG1 used to test the adherence of the program to RI1 requirements are shown in Figure 10.3.

After designing the required test cases, we execute them automatically on PRG1 to investigate the adherence of the mentioned PLC program to the RI1 requirement. As can be observed in Figure 10.4, all test cases have been exe-

Table 10.5: EARS Requirements examples obtained from the experiment and the resulting concretized EARS requirements.

Requirements	**EARS Requirements**	**Concretized EARS Requirements**
RI1	The <user account system> shall <identify the user> If <the user is not identified> then <user account system> shall <alert>	if <uniqueID=FALSE> then <UniqueUserAccount> shall <Result_Unique=FALSE>
RI2	When <malware is detected> the <system> shall <warn the user>	When <NormalActivity $\neq$ MaliciousActivity> the <MalwareDetection> shall <MalwareDetected=TRUE>
RI3	When <the device is authorised> the <system> shall <grant access to the device>	When <found=TRUE> the <SearchID> shall <ConnectionAllowed=TRUE>

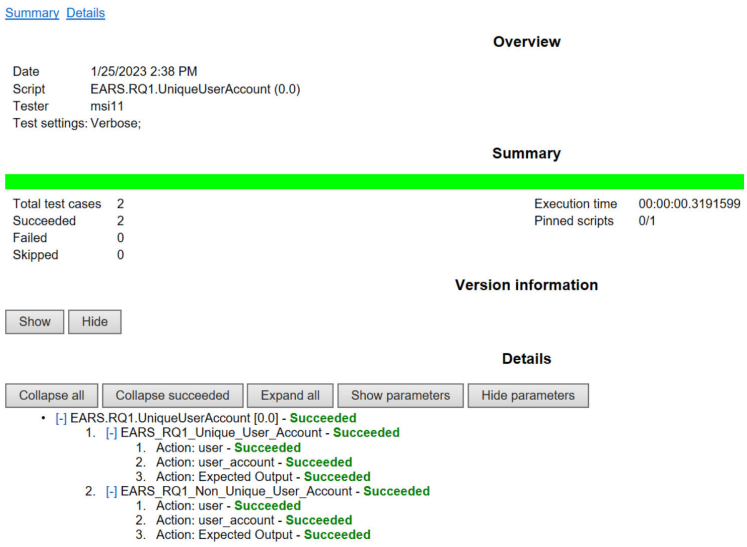

Figure 10.4: Test execution results for PRG1 PLC program based on the EARS-based generated test cases for RI1

cuted in 0.3 seconds. All executed test cases have successfully passed on the PRG1 program.

Test Results of *PRG2*

The PLC program we use for executing the generated test cases for *RI2* in Table 10.1 is named PRG2. This program is shown as a black-box malware detection system in the PLC environment that can be used for investigating the context of RI2. PRG2 consists of the following interfaces: two input signals named *MaliciousActivity* and *NormalActivity* as well as one output signal named *MalwareDetected*. When *MaliciousActivity* and *NormalActivity* signals have divergent information, the Malware Detection system is triggered, and the value of the *MalwareDetected* signal becomes True. An interface snippet of PRG2 is shown in Figure 10.5.

Considering the experiment results, we use the resulting EARS *Event-driven requirement* pattern as the most suited type of template for transforming the requirement from NL to EARS in the form shown in Table 10.5.

```
1  PROGRAM MalwareDetection
2  VAR
3      MalwareDetected: BOOL;
4      MaliciousActivity: BOOL;
5      NormalActivity: BOOL;
6  END_VAR
7
```

Figure 10.5: A snapshot showing the PRG2 PLC interface program written in the ST language in CODESYS IDE corresponding to the evaluation of the RI2 requirement.

Based on the developed EARS requirement for RI2 requirement, we generate two test cases for PRG2. Each test case consists of two test actions (*MaliciousActivity* and *NormalActivity*) that alter the value of the inputs, as well as one test action (*Expected Output* that compares the actual behaviour with the expected one. The first test case checks if a(*Malware is Detected*) while the second test case checks if a (*Malware is Not Detected*)

The generated test cases for PRG2 based on the RI2 requirement are then automatically executed using CODESYS Test Manager in 1.71 seconds. All developed test cases have successfully passed.

Test Results of *PRG3*

PRG3 is the PLC program used to execute the generated test cases for *RI3* in Table 10.1 (*"Only authorised devices are allowed to connect into the ICS network"*. This program consists of the following units: *1)* a database of authorised device IDs, which is implemented using an array of IDs, *2)* an input signal corresponding to the device ID that needs to be authorised, and *3)* a boolean output signal (i.e., *found*) which returns True in the case of the authorised device being allowed to connect given the ID is known. We show a snapshot of this PLC program in Figure 10.6.

As discussed in Section 10.6.1, different individuals transformed the NL requirement into the EARS requirement in different forms. We use the most common form developed by the participants to transform RI3 to an EARS *Event-Driven* syntax pattern in the following form shown in Table 10.5.

Based on the aforementioned EARS requirement for RI3, we developed 2 test cases for *Successful Authorization* and *Unsuccessful Authorization*. Each

```
1  PROGRAM SearchID
2  VAR
3    id_to_find : INT := 111;
4    found : BOOL;
5    array_of_ids : ARRAY[0..9] OF INT :=
6  [000,111,222,333,444,555,666,777,888,999];
7    i : INT;
8  END_VAR
```

Figure 10.6: A snapshot showing the PRG3 PLC program written in the ST language in CODESYS IDE corresponding to the evaluation of the RI3 requirement.

developed test case consists of two actions, including the provision of a *new Input ID* and *Comparing the actual output with the expected output*. The generated test cases have been automatically executed on PRG3 using CODESYS Test Manager in 1.14 seconds. Both test cases have successfully passed after being executed on the PRG3 PLC program.

10.7 EARS-based Testing in Real-world Industrial Settings

To expand our investigation of the applicability and efficiency of PLC testing using EARS patterns in real-world industrial settings (RQ4), in this section, we extend our evaluation by including a real-world PLC program that is being used in the context of crane supervision by a large automation company in Sweden. To be more specific, we compare the EARS requirement-based test cases with real-world test scripts that are being used for PLC testing by industry. We believe the conduction of this comparison can reveal hidden facts about the applicability and efficiency of using EARS-based testing versus the current real-world PLC testing in the industry.

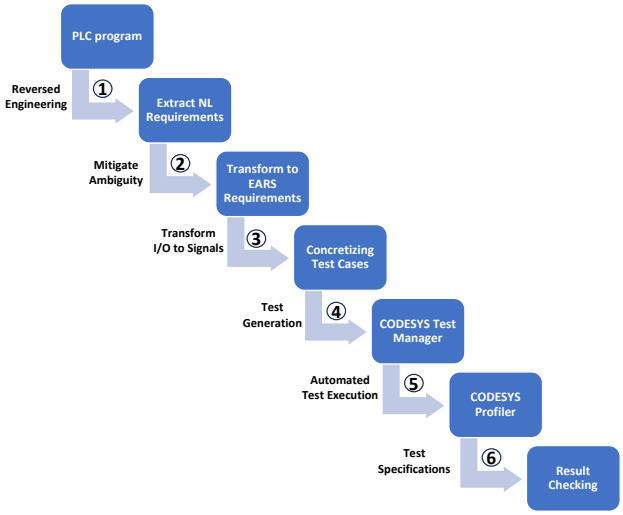

Figure 10.7: The proposed EARS-based testing method for real-world industrial PLC testing

10.7.1 Methodology for EARS-based testing in real-world industrial settings

The methodology we propose for using EARS-based testing in real-world industrial settings consists of six steps and is shown in Figure 10.7. The first step is to use reversed engineering to extract the functional requirements from the real-world PLC program (step 1 in Figure 10.7). The next step is to transform the NL requirements into EARS requirements to mitigate ambiguity and increase the clarity of the extracted requirements for the tester (Step 2 in Figure 10.7). As the next step, the EARS requirements need to be concretized for facilitating the test generation by converting the Inputs/Outputs (I/O) into signals (step 3 in Figure 10.7). After having the concretized test cases, it is time to generate test cases via the pre-defined test *Actions* inside the CODESYS Test Manager tool (step 4 in Figure 10.7). The next step is to automatically execute the test cases on the PLC program using the CODESYS Test Manager tool (step 5 in Figure 10.7). The final step in this methodology is to enhance the generated test specifications of CODESYS Test Manager by measuring the

code coverage using the CODESYS Profiler tool and checking the results (Step 6 in Figure 10.7).

10.7.2 Real-world Industrial PLC Program

In this section, We start by introducing the included real-world PLC program by defining its purpose and functionality. Then we analyze the industrial test script of this PLC program and compare it to our proposed EARS-based testing approach.

The included real-world PLC program in this work is called *CraneNumberCheck* and is shown in Figure 10.8. This PLC program serves the purpose of checking the match/mismatch of two crane numbers as part of a crane supervision program. As can be observed in Figure 10.8, this PLC program is developed in ST language and is composed of two input variables which represent the crane numbers and are called *Crane_1* and *Crane_2* (Lines 2-5 in upper box in Figure 10.8). Moreover, this PLC program consists of two output variables called *Matched_Crane_No* and *out_Safe_Crane_No* (Lines 6-9). The first one checks if the crane numbers match and are not empty *Word*, whereas the latter checks if the crane number is safe and if crane numbers mismatch, this word is set to an empty word.

As can be seen in the bottom box in Figure 10.8, the functional logic of the PLC program consists of two main parts and works as follows. In the first part, the *Matched_Crane_No* is set to True if the crane numbers are equal and *Crane_1* is not an empty *Word* (Lines 1-3 in the bottom box of Figure 10.8). The second part of the program's logic checks whether the crane numbers are matched or not. In case of success, the program returns the safe crane number, otherwise, it returns an empty *Word* (Lines 5-9 in the bottom box in Figure 10.8).

10.7.3 Industrial Testing of the Real-world Industrial PLC Program

The current testing process of the *CraneNumberCheck* PLC program in industry is handled by manually developing a counterpart testing *POU* in ST language. Part of the real-world industrial test script that is used for testing this PLC program is shown in Figure 10.11. As can be observed in Figure 10.11,

```
1  FUNCTION_BLOCK CraneNumberCheck
2  VAR_INPUT
3      Crane_1: WORD;
4      Crane_2: WORD;
5  END_VAR
6  VAR_OUTPUT
7      Matched_Crane_No: BOOL;
8      out_Safe_Crane_No: WORD;
9  END_VAR
10 VAR
11     EmptyWord: WORD;
12 END_VAR
```

```
1  Matched_Crane_No :=
2      (Crane_1 = Crane_2)
3      AND (Crane_1 <> EmptyWord);
4
5  IF Matched_Crane_No THEN
6      out_Safe_Crane_No := Crane_1;
7  ELSE
8      out_Safe_Crane_No := EmptyWord;
9  END_IF
```

Figure 10.8: A snapshot showing the *CraneNumberCheck* PLC program as a real-world industrial case study in the context of port crane supervision program

the industrial test script consists of several main steps. It starts with the initialization of a puls starting block as a trigger for starting the testing process, followed by an *IF* condition for enabling the test cases one by one (Lines 1-6 in Figure 10.11). The next step is the initialization of variables for test control (Lines 7-16 in Figure 10.11). After setting up the initialization, the next step is to define the main testing process which includes setting up a delay between test steps and the pulse generator followed by setting up a timer function that simulates the cyclic execution behaviour of PLC programs (Lines 18-26 in Figure 10.11). The rest of the testing process of *CraneNumberCheck* PLC program consists of unit test cases that define inputs and expected output.

10.7.4 Results of EARS-based Testing of a Real-world Industrial PLC Program

In this section, we use the proposed EARS-based testing methodology (refer to Figure 10.7) for testing the *CraneNumberCheck* PLC program as a real-world industrial case study. The first step is to reverse engineer the PLC program to

extract the functional NL requirements (Step 1 in Figure 10.7). The extracted NL requirements for this PLC program are shown in Table 10.6. This table also includes the used EARS pattern and the EARS version of each requirement which is described as step 2 of our proposed methodology in Figure 10.7. It is worth mentioning that all these functional NL requirements are validated by a team of experienced PLC engineers in a big automation company in Sweden.

As it can be observed in RQ2 and RQ3 rows of Table 10.6, the extracted functional requirements in NL can sometimes become complicated and hard to follow for the developers while their EARS version on the "Requirement in EARS" column are modularised and much easier to comprehend for the PLC program developer/testers. Moreover, we can observe that one complicated NL requirement can break into several smaller EARS requirements which also can increase the readability of the requirements.

After having the functional requirements in EARS syntax, we take the next step of our methodology which is concretizing the EARS requirements for generating unit test cases (step 3 in Figure 10.7). The procedure of concretizing the EARS requirements for PLC testing is simple and works as follows. Each I/O inside the requirement is transformed into a signal which can facilitate the test generation process as the next step. The concretized version of each EARS requirement for *CraneNumberCheck* PLC program is shown in Table 10.7.

The next step in testing the *CraneNumberCheck* PLC program based on the proposed testing approach is to generate test cases based on the concretized EARS requirements in the step before (Step 4 in Figure 10.7). To this end, first, we need to instantiate the *CraneNumberCheck* PLC program as a function block inside the main PLC program. A snippet of the function block we instantiated for *CraneNumberCheck* PLC program can be observed in Figure 10.9. As the next step, we used CODESYS Test Manager to design the test cases using the pre-defined *Test Actions* of this tool. After automatic execution of test cases on the *CraneNumberCheck* PLC program and using the CODESYS Profiler tool for measuring code coverage (Steps 5,6 in Figure 10.7), we gathered the following results. All the designed test cases with a timeout budget of 1 second have been successfully passed within 12 seconds on the PLC program under test. Moreover, the automatic test execution based on the proposed EARS-based PLC testing method for real-world industrial PLC programs achieved 100% code coverage on *CraneNumberCheck* PLC program based on the CODESYS Profiler report. A snippet of gathered full code cov-

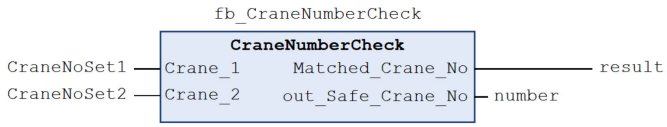

Figure 10.9: A snapshot showing the function block instantiation of *CraneNumberCheck* POU inside the main PLC program to prepare it for testing

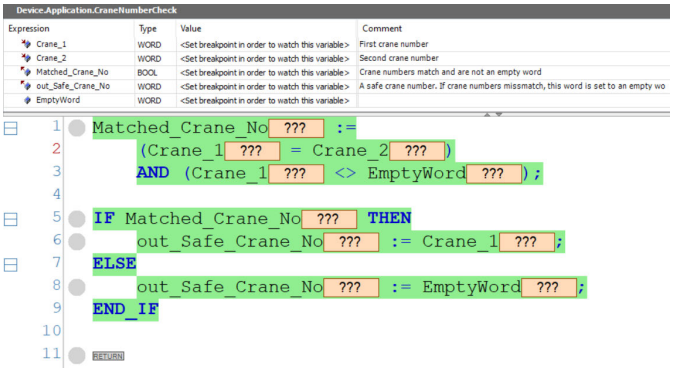

Figure 10.10: A snapshot showing the CODESYS Profiler report on gathered full coverage for *CraneNumberCheck* PLC program using the proposed EARS-based method (refer to Figure 10.7)

erage after testing the *CraneNumberCheck* PLC program is shown in Figure 10.10. As it can be observed in Figure 10.10, all the covered code branches after executing EARS-based test cases have been marked with green colour. The gathered results promise an acceptable level of applicability and efficiency of the proposed EARS-based testing method in the context of PLC programming, however, more investigation by applying this method to more complicated PLC programs needs to be done to validate the generalizability of this claim.

No	Functional Requirements:	EARS Pattern	Requirement in EARS
RQ1	"The function block shall accept two crane numbers (Crane_1 and Crane_2) as input parameters. (Input Requirements)"	Ubiquitous requirement (U)	The system shall accept two crane numbers (Crane_1 and Crane_2) as input parameters.
RQ2	"The function block shall provide two output variables: Matched_Crane_No: This variable indicates whether the crane numbers match and are not an empty word. out_Safe_Crane_No: If the crane numbers match, this variable shall store a safe crane number. If the crane numbers do not match, it shall be set to an empty word. (Output Requirements)"	Unwanted behaviours (UB)/State-driven requirements (SD)	UB: IF the crane numbers match and are not an empty word, THEN the system shall set Matched_Crane_No to true. SD: WHERE the crane numbers match, the system shall set out_Safe_Crane_No to a safe crane number. SD: IF the crane numbers do not match, THEN the system shall set out_Safe_Crane_No to an empty word.
RQ3	"The function block shall implement the following logic: Matched_Crane_No shall be true if Crane_1 is equal to Crane_2 and both are not empty words. If Matched_Crane_No is true, out_Safe_Crane_No shall be set to Crane_1. If Matched_Crane_No is false, out_Safe_Crane_No shall be set to an empty word. (Logic)"	State-driven requirement (SD)	WHERE the function block is active, the system shall: - Set Matched_Crane_No to true IF Crane_1 is equal to Crane_2 and both are not empty words. - Set Matched_Crane_No to false IF Crane_1 is not equal to Crane_2 OR either of them is an empty word. - IF Matched_Crane_No is true, THEN set out_Safe_Crane_No to Crane_1. - IF Matched_Crane_No is false, THEN set out_Safe_Crane_No to an empty word.
RQ4	"The function block expects that an empty word is a valid condition for not matching crane numbers. (Constraints)"	Ubiquitous requirement (U)	The system shall consider an empty word as a valid condition for not matching crane numbers.

Table 10.6: Industry-validated functional requirements for *CraneNumberCheck* PLC program (refer to Figure 10.8) in both NL and EARS versions

10.7.5 EARS-based Testing vs Manual PLC Testing in Industry

Comparing the overall current manual testing process of *CraneNumberCheck* PLC program in the industry versus the proposed EARS-based testing mechanism reveals several facts including:

1. **Need of domain-specific knowledge.** One needs to have a good understanding of one of the IEC61131-3 programming languages to be able to develop test cases for the PLC program in the current industrial approach. Moreover, the manual tester needs a testing background and

Requirement in EARS	Concretized Requirement
The system shall accept two crane numbers (Crane_1 and Crane_2) as input parameters.	if <Crane_1 & Crane_2 Exist=FALSE>then <SystemAcceptance>shall <Acceptance=FALSE>
UB: IF the crane numbers match and are not an empty word, THEN the system shall set Matched_Crane_No to true. SD: WHERE the crane numbers match, the system shall set out_Safe_Crane_No to a safe crane number. SD: IF the crane numbers do not match, THEN the system shall set out_Safe_Crane_No to an empty word.	UB: if <Crane_1 = Crane_2 & Crane_1 & Crane_2 $\neq$ Empty>then <Matched_Crane_No>shall <Matched_Crane_No=TRUE> SD: WHERE <Crane_1 = Crane_2>the <System>shall <out_Safe_Crane_No=SafeCraneNumber> SD: IF <Crane_1 $\neq$ Crane_2>THEN <System>shall <out_Safe_Crane_No=EmptyWord>
The function block shall implement the following logic: Matched_Crane_No shall be true if Crane_1 is equal to Crane_2 and both are not empty words. If Matched_Crane_No is true, out_Safe_Crane_No shall be set to Crane_1. If Matched_Crane_No is false, out_Safe_Crane_No shall be set to an empty word.	WHERE the function block is active, the system shall: - Set Matched_Crane_No to true IF Crane_1 is equal to Crane_2 and both are not empty words. - Set Matched_Crane_No to false IF Crane_1 is not equal to Crane_2 OR either of them is an empty word. - IF Matched_Crane_No is true, THEN set out_Safe_Crane_No to Crane_1. - IF Matched_Crane_No is false, THEN set out_Safe_Crane_No to an empty word.
"The system shall consider an empty word as a valid condition for not matching crane numbers."	if <Crane_1 $\neq$ Crane_2>then <SystemAcceptance>shall <Acceptance=Empty Word>

Table 10.7: The concretized requirements of *CraneNumberCheck* PLC program (refer to Figure 10.8) which are generated based on the requirements in EARS syntax.

engineering experience to implement and connect all testing units properly. On the other hand, testing the PLC programs with the proposed mechanism using CODESYS Test Manager does not demand any deep knowledge of specific programming language and can be handled easily using *Test Actions*.

2. **Efficiency.** In the case of a simple PLC program such as *CraneNumberCheck* which consists of 25 Lines of Code (LOC), the test script consists of 119 LOC which shows a difference in efficiency. On the other hand, the proposed EARS-based testing approach only consists of 26 test actions, which use all the powerful features of CODESYS IDE.

3. **Manual overhead and complexity.** The current testing process for PLC programs in the industry is highly complex, with significant manual intervention. Specifically, many features already present in the CODESYS IDE, such as cyclic execution, delay, test control process, and test start trigger, are being recreated manually. This redundancy exacerbates the complexity, especially with more intricate PLC programs. Conversely,

the proposed EARS-based testing approach simplifies this by requiring the manual definition of only the inputs and expected outputs. All other features are readily accessible through the user-friendly GUI of the CODESYS Test Manager tool. Additionally, the availability of predefined test actions within the Test Manager tool enhances the use of CODESYS's features and automation capabilities for PLC testers.

4. **Test specifications.**The existing manual testing process in the industry offers testers limited information, providing only the outcomes of passed or failed test cases. In contrast, the proposed EARS-based testing approach utilizes both CODESYS Test Manager and CODESYS Profiler to provide a comprehensive set of test specifications. These include additional details like test execution time, coverage reports, outcomes of individual test actions, test verdicts, and more.

5. **Ambiguity and clarity of functional requirements.** After reviewing a limited set of requirements gathered from the industry, it became apparent that the current functional requirements are predominantly at the system level, lacking specificity for individual code branches. Additionally, the complexity of industrial testing processes relies heavily on the tester's expertise. In contrast, the proposed EARS-based approach reduces the vagueness of requirements and encompasses both unit and system-level testing, potentially leading to a more thorough testing procedure. Furthermore, this approach yields requirements and test cases that are straightforward and comprehensible, facilitating understanding among all stakeholders, including testers, managers, and clients.

10.7.6 Limitations of the Study and Threats to Validity

External Validity. All of our subjects are individuals who have limited experience with EARS. Furthermore, because these practitioners have experience in requirements engineering, we see no reason the use of professionals with deep knowledge of EARS in our study would yield a completely different result. Professionals with experience in EARS would intuitively write better requirements than the ones written by our subjects. Our study has focused on three relatively brief with reduced complexity, but these requirements represent relevant samples they would encounter in practice. We have used the CODESYS

```
(* Initialization, puls starting block *)
_RTRIG1(CLK:= in_Start, Q=> );
(* Enable testing case by case *)
IF (in_Start OR _RTRIG1.Q) AND in_StartC THEN
    in_StartC := FALSE;
END_IF
(* Initialization of variables for test control *)
IF (_RTRIG1.Q) OR (in_StartC AND (step >= EnumValueSTest_TestDone)) THEN
    FOR i := 1 TO 7 BY 1 DO
        out_Status.Case_Ok[i] := FALSE;
    END_FOR
    out_ActCase := 0;
    out_All_Ok := FALSE;
    out_Status.Done := FALSE;
    step := 1;
END_IF

(* Test process *)
IF in_Start OR  in_StartC THEN
    (* Delay between test steps and pulse generator *)
    (* Setting of timer, every case will be long minimally 250ms *)
    _TON1(IN:= step = stepOld, PT:= timerTime, Q=>, ET=> );

    (* Default state needed for all test cases --- Area for default setting *)
    craneNoSet1 := 0;
    craneNoSet2 := 0;

    (* Test procedure *)
    CASE step OF
        1:  (* Set  for case 1 *)
            (* All inputs are same as in area for default setting *)
            out_ActCase := 1;

        2:  (* Set  for case 2 *)
            (* All inputs are same as in area for default setting except input -> Crane_1 *)
            craneNoSet1 := 3;
            out ActCase := 2;
```

Figure 10.11: A snippet showing part of the real-world test script for testing the *CraneNumberCheck* PLC program in the current industry.

tool for automated test creation and execution. There are many tools for developing and executing tests, and these may give different results. Nevertheless, CODESYS is one of the most used development environments for PLCs, and its output in tests is similar to the output produced by other tools.

Internal Validity. All subjects were assigned to perform the experiment at the same time. This was dictated by the way the experiment was organized, with a presentation followed by practical work. Subjects without sufficient knowledge of EARS may affect the final result. To avoid this problem, the session was structured to follow the corresponding EARS lesson. Another threat to internal Validity could arise from using unclear objectives given to the subjects. To address this, we tested the material ourselves.

Construct Validity. Capturing the challenges of requirements engineering and testing is a difficult problem. We rely on human feedback by using a think-out-loud method that gives a rough measure of the challenges encountered.

Conclusion Validity. The results of this study are based on an experiment using 10 participants and three requirements. For each requirement, all participants

performed the study, which is a relatively small number of subjects. Nevertheless, this was sufficient to obtain various results showing an effect between the modelling of these different types of requirements.

10.8 Related Work

Mavin and Wilkinson [15] reflected on the ten years of EARS [8] and shared some lessons learned in their review paper. For example, they have discovered that users of EARS manage to author more useful draft requirements as they incrementally work to find the appropriate EARS pattern. They recommend that new engineers write several requirements and seek expert review with the application of EARS being more useful if one can apply the following activities: training, thinking, semantics, syntax, and review. In our study, we confirm some of these results even if we do not cover all these activities stated.

Mavin et al. [16] report on the understanding of four experienced EARS practitioners and their reflections on their experiences of applying EARS in different projects and domains over six years. They report the following EARS-specific lessons learned: training should be short, use EARS with or without a tool, use coaching to embed learning, challenge the EARS Patterns, and question if the EARS clauses are necessary and sufficient.

Mäntylä et al. [17] performed a controlled experiment on test case development and requirement review and the effects of time pressure. They saw no statistically significant evidence that time pressure would lower effectiveness or provoke negative influences on motivation, frustration, or performance.

Dalpiaz et al. [18] investigated the adequateness, completeness, and correctness of use cases and user stories for the manual creation of a static conceptual model. They performed a controlled experiment with 118 subjects, and their results show that for creating conceptual models, user stories work better than use cases. Furthermore, user story repetitions and conciseness contribute to these results. However, as we aim with our study, more evidence needs to be provided regarding the aspects that must be considered when selecting and using a modelling and requirement notation.

Weninger et al. [19] report the results of a controlled experiment in which they compared two approaches for defining restricted use case requirements from multiple perspectives, including misuse, understandability, and restric-

tiveness. Their results indicate the usefulness of the restricted use case modelling approach.

The paper by Levi Lucio et al. [20] introduces the EARS-CTRL tool, an editor built on MPS, designed to aid in the crafting and analysis of EARS requirements for controllers. This editor inherently ensures well-formedness, offering a structured method and suggesting relevant terms from a glossary during the editing process. The paper discusses automated checks for the feasibility of requirements, utilizing a controller synthesis tool, and the generation of synchronous dataflow diagrams for verified requirements. While it recognizes the challenges in representing complex states, the research emphasizes the importance of closing the gap between natural language requirements articulated in EARS and formal specifications, thereby enhancing the automation of requirement analysis and synthesis within an industrial setting.

10.9 CONCLUSIONS AND FUTURE WORK

In this paper, we have conducted an experiment in requirements engineering and testing using EARS notation for PLC systems. In the requirement engineering part of our experiment, we found out that most participants preferred the EARS ubiquitous pattern for transforming the RI1 requirement from NL to the EARS syntax, whereas the unwanted behaviour and event-driven patterns were the most popular types for RI2 and RI3 requirement transformations. It was observed that different individuals used different EARS patterns for transforming the same requirement based on their personal interpretation, which shows an acceptable level of flexibility in EARS syntax. In the testing part of our experiment, we assessed the use of EARS patterns for PLC testing in two phases. Initially, we executed EARS-based test cases on three PLC programs written in the ST language, which were developed based on the requirements included in our study. Subsequently, we introduced an EARS-based testing methodology to real-world industrial PLC programs. The results from these tests and the subsequent comparison with traditional PLC testing methods indicate that EARS-generated requirement-based test cases for PLC programs are effective and offer an accessible means for PLC testers to express test specifications.

In future work, we want to investigate the applicability of using EARS in PLC requirement engineering on other levels of testing and by including more

PLC programs. Inspection of the impact of choosing different EARS templates for describing the requirements over the quality of the generated test cases can be another future direction of our work. Moreover, we want to automate our solution and generate test cases from the created EARS requirements based on existing functional and non-functional requirements.

Acknowledgment

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

Statements and Declarations

Competing Interests. The authors declare that they have no conflict of interest.

Funding Information. EU's H2020 research and innovation program under grant agreement No 957212 and from Vinnova through the SmartDelta project.

Author contribution. Mikael Ebrahimi Salari is the main driver and contributor of the paper while the rest of the authors contributed to the methodology and provided valuable feedback.

Data Availability Statement Synthetic data and also data from industry.

Research Involving Human and /or Animals. Not Applicable.

Informed Consent. Informed consent was obtained from all individual participants included in the study.

Bibliography

[1] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. Control System Devices: Architectures and Supply Channels Overview. In *Sandia Report SAND2010-5183*. Sandia National Laboratories, 2010.

[2] CENELEC. 50128: Railway Application–Communications, Signaling and Processing Systems–Software for Railway Control and Protection Systems. In *Standard Report*. 2001.

[3] 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. In *International Journal on Software Tools for Technology Transfer*, volume 18, pages 335–353. Springer, 2014.

[4] Yi-Chen Wu and Chin-Feng Fan. Automatic Test Case Generation for Structural Testing of Function Block Diagrams. In *Information and Software Technology*, volume 56. Elsevier, 2014.

[5] E. Jee, J. Yoo, S. Cha, and D. Bae. A data flow-based structural testing technique for FBD programs. In *Information and Software Technology*, volume 51, pages 1131–1139. Elsevier, 2009.

[6] Kivanc Doganay, Markus Bohlin, and Ola Sellin. Search Based Testing of Embedded Systems Implemented in IEC 61131-3: An Industrial Case Study. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 425–432. IEEE, 2013.

[7] Vahid Garousi and Junji Zhi. A Survey of Software Testing Practices in Canada. In *Journal of Systems and Software*, volume 86, pages 1354–1376. Elsevier, (2013).

[8] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (ears). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322. IEEE, 2009.

[9] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, and Cristina Seceleanu. An experiment in requirements engineering and testing using ears notation for plc systems. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 10–17. IEEE, 2023.

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

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

[12] Dag H Hanssen. *Programmable logic controllers: a practical approach to IEC 61131-3 using CODESYS*. John Wiley & Sons, 2015.

[13] Virginia Braun and Victoria Clarke. *Thematic analysis.* American Psychological Association, 2012.

[14] Flemström Daniel, Enoiu Eduard, Azal Wasif, Sundmark Daniel, Gustafsson Thomas, and Kobetski Avenir. From natural language requirements to passive test cases using guarded assertions. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 470–481. IEEE, 2018.

[15] Alistair Mavin Mav and Philip Wilkinson. Ten years of ears. *IEEE Software*, 36(5):10–14, 2019.

[16] Alistair Mavin, Philip Wilksinson, Sarah Gregory, and Eero Uusitalo. Listens learned (8 lessons learned applying ears). In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 276–282. IEEE, 2016.

[17] Mika V Mäntylä, Kai Petersen, Timo OA Lehtinen, and Casper Lassenius. Time pressure: a controlled experiment of test case development and requirements review. In *Proceedings of the 36th International Conference on Software Engineering*, pages 83–94, 2014.

[18] Fabiano Dalpiaz and Arnon Sturm. Conceptualizing requirements using user stories and use cases: a controlled experiment. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 221–238. Springer, 2020.

[19] Markus Weninger, Paul Grünbacher, Huihui Zhang, Tao Yue, and Shaukat Ali. Tool support for restricted use case specification: Findings from a controlled experiment. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 21–30. IEEE, 2018.

[20] Levi Lúcio, Salman Rahman, Chih-Hong Cheng, and Alistair Mavin. Just formal enough? automated analysis of ears requirements. In *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings 9*, pages 427–434. Springer, 2017.

Chapter 11

Paper D: 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, Filip Sebek
Published in the 30th Asia-Pacific Software Engineering Conference (APSEC 2023), Software Engineering In Practice (SEIP) Track.

11.1 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.

11.2 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

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

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.

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 bring-

ing automated test case generation for Python to CODESYS, hence enabling automated verification of correct functionality of FBD PLC programs.
The paper is organized as follows. Section 11.3 briefly overviews the preliminaries on PLC, IEC 61131-3 standard, and Python. Section 11.4 describes how the proposed automated translation framework works. Section 11.5 explains the procedure of automated validation of the translated code using metaheuristic algorithms. Section 11.6 overviews the results of this study. Section 11.7 presents related work, whereas Section 11.8 concludes the paper and outlines future research directions.

11.3 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.

11.3.1 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 11.1. The widely accepted CODESYS IDE, aligned with IEC 61131-3, 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.

11.3.2 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,

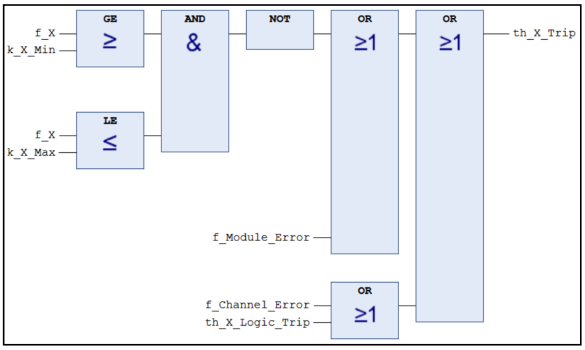

Figure 11.1: A snippet of an FBD program for controlling the temperature in a nuclear plant (PRG9)

improving code coverage and test suite quality [8]. It also integrates advanced algorithms like Dynamic Search-Based Software Testing (DSBST) and Evolutionary Testing (ET). Additionally, Pynguin employs *DYNAMOSA* [13] 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.

11.3.3 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).

[2]https://bit.ly/44uSUYz

11.3.4 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 [14]. 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 [15]. A snippet of part of a PLCopen XML tree for the PRG9 FBD program is shown in Listing 11.3.

11.3.5 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 [16]. However, it also poses some challenges for testing and debugging PLC programs, as data flow relationships and reachable states need to be considered[4].

11.3.6 Data Types in IEC61131-3 and Python

Programming languages use diverse data types, shaped by their design goals and paradigms. This section contrasts IEC 61131-3's and Python's data types.

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

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.

11.4 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 11.2. 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 11.2). Subsequently, PyLC uses this extracted information to automatically generate executable Python

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

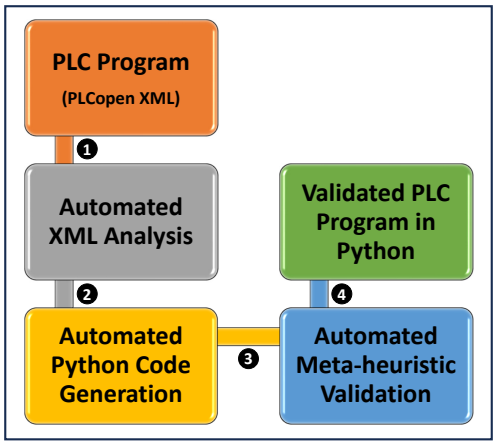

Figure 11.2: An Overview of the PyLC Framework, the Proposed Automated Translation Mechanism for Translating a PLC Program into Python Code and Validating the Translation Automatically.

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 11.2). Next, PyLC employs the meta-heuristic automated unit testing techniques from Pynguin tool [8] to validate the translation (Step 3 in Figure 11.2). 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 11.2).

The detailed step-by-step PyLC workflow follows.

11.4.1 PyLC Translation Workflow

The translation workflow of PyLC, as demonstrated in Figure 11.3, consists of five main stages spanning both the PLC and Python environments. In this section, we describe each step of the PyLC translation process.

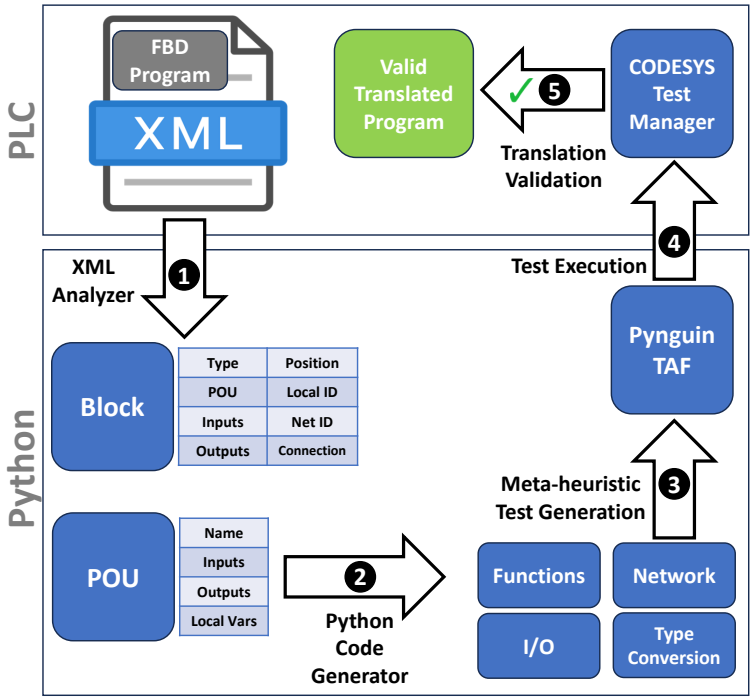

Figure 11.3: The Automated Translation Work Flow (TWF) Used in PyLC Framework for Translating a PLC Program into Python with the assistance of the Pynguin Test Automation Framework (TAF).

Step 1 - XML Analyzer

The XML Analyzer module of PyLC (Step 1 in Figure 11.3) 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 11.1. This module leverages the *ElementTree XML* [6] module of Python and classifies all the extracted

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

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 11.1, 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 Parameters*, *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 11.1: 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')
```

```
27          Initialize empty list 'B_input_vars'
28          @For{each 'input_var' in block.findall('.//plcopen:inputVariables/
                plcopen:variable', ns)}:@
29              Get 'input_local_id' from 'refLocalId' attribute of 'input_var
                    '
30              Append 'input_local_id' to 'B_input_vars'
31          @EndFor@
32          % ... (similar steps for 'B_var_formal_param', 'B_conn_point_in',
                'B_conn_ref_local_id')
33          Write information about block into 'file.py' using the generated
                variables
34      @EndFor@
35  @EndFor@
36  Write information about POU into 'file.py'
37  Close 'file.py'
```

Step 2 - Python Code Generator

The Python Code Generator unit of the PyLC workflow (Step 2 in Figure 11.3) 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 11.2.

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 11.2). 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 11.2). 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 11.2) and converting the variable data types

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 11.1: The list of the supported FBD Blocks in the PyLC automated translation framework

from PLC to Python (Lines 60-62 in Listing 11.2).

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 11.2). 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 11.3). The list of the supported IEC61131-3 standard FBD Blocks based on their category in the PyLC framework is shown in Table 11.1.

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 11.3). 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 11.3) 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 11.3) and converts it to either equivalent or similar data type in Python (Lines 60-61 in Listing 11.2). 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 11.4.2 (Lines 12-28 in Listing 11.5).

Listing 11.2: The Abstracted Pseudo-code for Python Code Generation Module of PyLC

```
1 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)
```

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 11.3). 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 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 11.4.

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 11.3). 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.

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 11.3). 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 invalid.

11.4.2 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 11.1. 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 11.3.

Listing 11.3: Part of the PLC Open XML Tree of PRG9 PLC Program

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <project xmlns="http://www.plcopen.org/xml/tc6_0200">
3   <fileHeader companyName="" productName="CODESYS" productVersion=
4   "CODESYS␣V3.5␣SP16" creationDateTime="2023-08-14T13:43:53.0957274" />
```

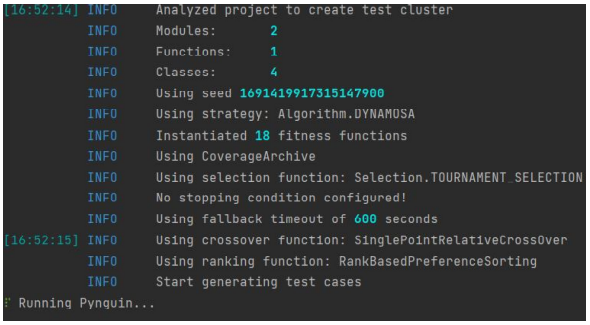

Figure 11.4: A Snippet of The Pynguin Test Generator Processing a Translated PLC program into Python using the PyLC Automated Translation Framework

```
<contentHeader name="" modificationDateTime="2023-08-14T13
    :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 11.3). Part of the results of applying the PyLC XML Analyzer module on PRG9 is shown in Listing 11.4. As shown in the Listing 11.4, the extracted information from the XML tree using the PyLC XML analyzer module is classified based on the *Blocks* (B1-B7) and *POU*.

Listing 11.4: 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 11.3). We show part of the resulting generated Python code for PRG9 using the PyLC translation framework in Listing 11.5.

Listing 11.5: 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():
    def str_to_bool(s):
        return s.lower() in ('true', 't', '1')
    def str_to_int(s):
        try:
            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 11.5, 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 11.5). 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 11.5). 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 11.5).

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 11.5).

11.5 Automated Validation of The Translated Code using Meta-heuristic Algorithms

In this study, we use the DYNAMOSA algorithm [17] as the selected meta-heuristic algorithm for validating the correctness of the translated PLC program into Python.

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 [17]. This empowers Pynguin to efficiently create well-balanced test cases.

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 11.3).

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 11.3).

11.6 Results

11.6.1 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 Se-

lection as our selection function, Single Point Relative Crossover for crossover, and Rank-Based Preference Sorting for ranking.

11.6.2 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 11.2. 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 11.2, 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 11.2, we can draw several conclusions. First, the FBD programs selected for translation encompass a variety of FBD block types, as detailed in Section 11.2. 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 automated 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 within industrial automation but also offers a systematic approach to bridge the gap between PLC programming languages and general-purpose languages like Python.

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 11.2: Information Regarding the Translated PLC Programs (PRG) in FBD language into Python using PyLC

11.6.3 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 11.3. 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 11.3, 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, Pyn-

guin 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 behaviour. Iteration counts vary 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

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 11.3: Information Regarding Automated Testing of The Translated Real-world PLC Programs to Python using the Pynguin Tool

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 automated 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.

11.6.4 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 11.3.6, 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 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 11.3), 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.

11.7 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.

11.7.1 Program Transformation to Python for Enhanced Features and Tools

Peterson et al. [18] 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 [19]. 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" [20], 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" [20].

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.

11.7.2 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. [21] 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 mod-

els, demonstrated via a water treatment case study. The study by Tychalas et al. [22] 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 [23] and runtime verification [24]. He et al. [23] 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. [25] 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.

11.8 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 trans-

lation'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 automation companies, at the unit level.

In future work, we aim to conduct a more thorough examination of the scalability of PyLC, investigate the compatibility of timer blocks with Pynguin, as well as enhance the translation validation mechanism of PyLC with a Python static verifier.

Acknowledgment

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

Bibliography

[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] Jiayi Wu and Qingfu Zhang. Dynamosa: A dynamic multi-objective search algorithm for continuous optimization problems. In *Proc. of the 2018 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2018.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.