PyLC+: A Scalable Python Framework for Automated Translation and Testing of Industrial PLC Programs*

Mikael Ebrahimi Salari[1], Eduard Paul Enoiu[2], Alessio Bucaioni[3], Wasif Afzal[4], Cristina Seceleanu[5]
School of Innovation, Design, and Engineering (IDT)
Mälardalen University
Västerås, Sweden
[1]mikael.salari@mdu.se, [2]eduard.enoiu@mdu.se, [3]alessio.bucaioni@mdu.se, [4]wasif.afzal@mdu.se, [5]cristina.seceleanu@mdu.se

***Abstract*—As industrial PLC programs become more complex, automated testing and verification methods are needed to ensure their reliability and correctness. This paper presents PyLC+, a modular framework that translates PLC programs into Python, allowing for automated AI-driven test generation. PyLC+ builds upon our previous work, addressing limitations by adopting a class-based modular architecture that improves the tool's scalability, maintainability, and extensibility. This structural refinement eliminates reliance on nested functions, facilitating the translation of large-scale, real-world PLC programs while maintaining precise use of cyclic execution. Furthermore, PyLC+ introduces automated handling of *stateful* FBs, ensuring compliance with IEC 61131-3 execution semantics.**

Additionally, the tool proposes integrating LLM-driven test generation with search-based test generation to improve the efficiency and effectiveness of testing PLC software. We tested PyLC+ in a large-scale company developing train control systems, demonstrating its efficiency and effectiveness in handling complex industrial PLC programs.

***Index Terms*—PLC, FBD, Test Automation Framework, Python, AI, LLM, Search-based Testing, PyLC+**

I. Introduction

Industrial control systems (ICS) are essential to automated operations in manufacturing, energy, transportation, and utilities. At the core of ICS are Programmable Logic Controllers (PLC), which execute real-time control logic, processing inputs from sensors and producing outputs to actuators. IEC 61131-3 [1] is an international standard for PLC programming languages, defining five distinct paradigms out of which Function Block Diagram (FBD) is used in this paper. FBD is widely used due to its graphical representation, which simplifies the design of complex control logic using interconnected Function Blocks (FB). Testing such PLC programs relies heavily on domain expertise and manual effort, making it time-consuming, especially as PLC software becomes more complex [2].

Although manual methods are widespread, they seem to struggle to scale and adapt to the growing complexity of industrial PLC software [3]. Automated approaches such as Search-Based Testing (SBT) have emerged as practical solutions to systematically generate test suites by formulating testing as an optimization problem [4]. There is a need for more scalable testing approaches to address the challenges posed by the growing complexity of PLC programs.

To address the above challenges, this paper proposes the PyLC+ framework, an automated Python-based testing and translation solution designed explicitly for industrial PLC software. The PyLC+ framework is built upon our previous work [5] [6], and presents a hybrid testing method for PLC software, combining SBT with Large Language Model (LLM)-driven test generation to handle stateful logic, significantly improving test coverage and effectiveness compared to existing techniques. This study focuses only on the functional requirements of PLC programs, including timing-sensitive ones. Results from evaluating industrial PLC programs indicate that PyLC+ achieves high branch and mutation coverage, confirming its applicability and efficiency for safety-critical industrial PLC software.

II. Background

A. ICs and FBD Programs

As ICS evolves in scale and complexity, ensuring the correctness and reliability of PLC programs has become a major challenge. Even minor errors in the control logic can lead to significant consequences, such as operational disruptions, financial losses, or safety hazards [7], [8]. For instance, consider a PLC program implementing basic logic using FBD, which involves arithmetic operations, comparisons, and timers, as shown in Figure 1. This FBD program adds 3 to *nVar1*, compares the result with *nVar2*, and if the condition holds, it triggers an on-delay timer (TON) with a preset time of 12 ms. Once the timer elapses, the output *bVar1* is set to TRUE, ensuring a delayed activation based on the input conditions. This structure is useful for industrial automation, where precise timing and conditional logic are required.

In FBD programming, FBs are classified as stateful or *stateless* based on whether they retain internal memory across PLC scan cycles. *stateful blocks*, such as timers, counters, and

This work has received funding from the MATISSE project, an EU-funded initiative under Horizon Europe GA no. 101056674, and support from the SmartDelta project funded by Vinnova.

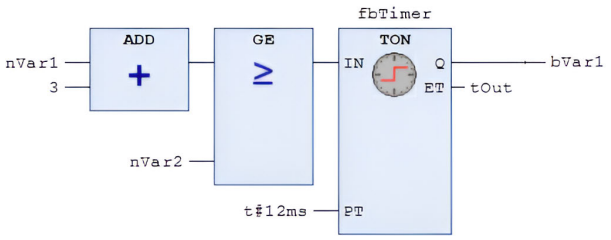

Fig. 1: A simple PLC program in FBD language representing a control logic using arithmetic operations, comparisons, and a timer function block

flip-flops, store data from previous cycles, making their outputs dependent on both current inputs and past execution history. These blocks are essential for implementing sequential logic and event-driven behaviors. In contrast, stateless blocks, like logical gates and arithmetic functions, produce outputs solely based on present inputs, without retaining any memory. This distinction is crucial in industrial automation, where stateful blocks enable time-dependent control logic, while stateless blocks ensure immediate responsiveness to input conditions.

B. Verification and Validation of PLC Programs

The verification and validation of PLC programs are crucial in mitigating risks by ensuring that control logic meets safety and performance requirements. Traditional testing approaches, including test generation and model checking, have been employed in industrial settings [9]. While rooted in domain expertise, testing of such systems is still time-consuming, error-prone, and infeasible for highly complex systems [10].

In recent years, AI and LLMs have emerged as tools for automating various software testing tasks. AI-based methods have demonstrated significant potential in automating test generation, analyzing system behavior, and identifying subtle faults [11], [12]. LLMs, with their ability to process and generate human-readable code, have demonstrated capabilities in understanding and generating programs directly from specifications [13], synthesizing code from existing codebases [14], and even repairing runtime errors [15]. Despite these advances, there remain several challenges to overcome. Existing AI-driven testing frameworks often lack modularity and scalability, limiting their adaptability to diverse industrial contexts [16], [17]. Additionally, the applicability and performance of LLMs in comparison to traditional techniques such as automated test generation and manual testing remain under-explored [18].

III. Research Gaps, Research Goal and Objectives

This section analyzes the existing Research Gaps (RGps) in the automated translation and testing of complex industrial PLC programs. Then, we outline this work's overall research goal, which will be investigated through several different research objectives developed to fill in the mentioned gaps. Then, we map the contributions of this work to each research objective later in Section VI when we discuss the gathered results.

A. Research Gaps

Despite advancements in automated testing of PLC programs, several challenges remain unaddressed. These include handling complex real-time behaviors [19], ensuring coverage of edge cases [20], integrating with legacy systems [21], and addressing the lack of standardized testing frameworks [3].

RGp1 - Limited Modularity and Scalability in PLC Translation Tools Existing PLC-to-Python translation frameworks. For instance, the initial version of PyLC [5] [6] relies on a nested function-based architecture, resulting in high code complexity, reduced maintainability, and limited scalability, especially when translating large industrial PLC programs.

RGp2 - Inadequate Handling of stateful FBs. Many current PLC testing frameworks fail to distinguish and manage stateful blocks properly, leading to incorrect simulation of IEC 61131-3 execution semantics and impacting the validity of test cases [22]. These limitations highlight the need for improved handling of stateful FBs in testing methodologies.

RGp3 - Lack of Efficient Automated AI-Driven and LLM-Based Test Generation for PLC Programs. While AI and LLM-based test generation methods have gained traction in software testing [4], their application in PLC testing remains unexplored, potentially limiting improvements in automation and efficiency in test case generation and validation.

RGp4 - Absence of Static Program Verification for Translated PLC Programs. Static verification of PLC-to-Python translated code is currently impractical due to architectural constraints [23]. In particular, Python-based formal verification tools (e.g., *Nagini* [24]) require structured class-based architectures, which are incompatible with existing nested-function-based PLC translation methods.

B. Research Goal and Contributions

The primary goal of this research is to develop and evaluate a modular, class-based PLC-to-Python automated translation framework that addresses the above gaps. To achieve this, our objectives are: i) Provide a valid and scalable translation architecture for complex industrial PLC programs using a class-based modular design, ii) Improve stateful block management to emulate PLC cyclic execution semantics, to ensure correct runtime behavior in translated Python programs, iii) Enable automated AI-driven test generation to serve the effectiveness (better coverage), efficiency, and validation for translated PLC programs, and iv) Enable future integration of static formal verification for translated PLC programs by ensuring architectural compatibility with Python verification tools such as *Nagini* [24].

The contributions of this study that address the overall goal by meeting the identified objectives are shown in Table I. These contributions collectively address the primary research goal of developing a modular, class-based PLC-to-Python translation framework by enhancing compatibility, scalability, and correctness, while enabling automated AI-driven test generation and future integration with formal verification tools.

Contribution	Objective Addressed
RCo1	**Expanded Compatibility** Support for the `PLCopen XML` and `.pou` file formats to meet real-world industrial needs.
RCo2	**Modular and Scalable Architecture** A class-based design that reduces code complexity, improves scalibility and streamlines the test case generation process.
RCo3	**Enhanced Large-Scale Support** Enables translation and simulation of complex real-world PLC programs, validated in industrial contexts (e.g., railway systems).
RCo4	**Emulation of PLC Cyclic Execution** Accurately simulates PLC runtime behavior in Python to ensure correct execution semantics.
RCo5	**LLM-driven Automated Test Generation** Integrates automated AI-driven test case generation using metaheuristic algorithms and LLMs to improve the translated programs' coverage, efficiency, and validation.
RCo6	**Automatic Detection and Handling of Stateful and Stateless Blocks** Automatically differentiates between the stateful and stateless FBs, ensuring adherence to the IEC 61131-3 standard's execution semantics.
RCo7	**Automatic Identification of the PLC Program State** Identifies the presence of stateful elements to optimize test generation strategies while keeping the state in memory among different execution cycles.
RCo8	**Enabling automated static verification compatibility** Added support for the future formal verification of the translated PLC code by meeting architectural requirements for valid state-of-the-art tools such as *Viper* and *Nagini*.
RCo9	**IDE-independent design**: Removes any dependencies on a specific PLC IDE, increasing applicability across diverse industrial PLC programs.

TABLE I: Contributions and Corresponding Objectives of This Study

IV. PyLC+: An Automated Modular PLC to Python Translation & Testing Framework

This section details the architecture of PyLC+, a framework for automatically translating PLC programs into Python and validating them using a hybrid AI-based testing approach. This new version improves the original PyLC framework by: i) transitioning from nested functions to a modularized class-based architecture for improved scalability and maintainability, ii) implementing AI-driven testing for better handling the stateful PLC programs, addressing the limitations of SBT in covering cyclic execution and persistent states, and iii) retaining SBT (via *Pynguin* [25]) for stateless PLC programs, ensuring effective functional verification where applicable. Figure 2 illustrates the tool's workflow, which consists of four steps: (1) Parsing the PLC program, (2) Generating modular Python code, (3) Test-case generation, and (4) Test execution and validation. In the following, we describe these steps and their corresponding modules in more detail.

A. PyLC+ PLC Parser Unit

The first step in PyLC+ involves extracting components from the FBD PLC program under translation imported as a *PLCopen* XML file (Step 1 in Figure 2). This process is handled by the POU Parser module of the PyLC+ tool (Step 2 in Figure 2 and involves i) Automatic extraction of the variables, including inputs, outputs, and local variables as well as their respective data types. ii) Automatic detection of the FBs and existing logical operations (e.g., AND, OR, GT) inside them. iii) Automatic detection of the stateful (e.g., TON, RS, COUNTER) and stateless (e.g., AND, XOR) components. iv) Mapping connections and establishing the functional and logical relations between FBs and variables to define execution dependencies.

The implementation of this Automatic XML Parser unit has been done by developing a dynamic Python script that can parse and process the XML tree data from either a PLCopen *.XML* or a *.POU* file. This script transforms this extracted data into an intermediate representation (*JSON*) stored in a new Python file. This structured data is the foundation for the subsequent translation into Python code.

B. PyLC+ Built-in IEC 61131-3 Block Factory

An improvement in PyLC+ is the shift to a modular class-based approach, replacing the previous nested function implementation. This block factory is a Python repository that provides a dynamic instantiation mechanism for FBs, improving the modularity of the translated code. This redesign brings several benefits, such as the encapsulation of the FBs. This means that each PLC block is implemented as a separate Python class with clear input/output definitions, and the PyLC+ Code Generator Unit automatically calls the class instance upon need in the next steps of translation. The repository of these blocks, which are all defined based on the IEC 61131-3 standard semantics, is called the PyLC+ built-in Block Factory.

C. PyLC+ Block Type Detector Unit

According to the IEC 61131-3 standard [1], FBD programs consist of two FB categories, including stateless and stateful FBs. stateless blocks (e.g., AND, OR, GT) are implemented as simple, side-effect-free functions that do not keep the internal state over the execution cycles, and their values should be reset after each execution cycle; however, the stateful blocks (e.g., TON, RS, COUNTER) are instantiated as persistent objects that retain their state across different execution cycles.

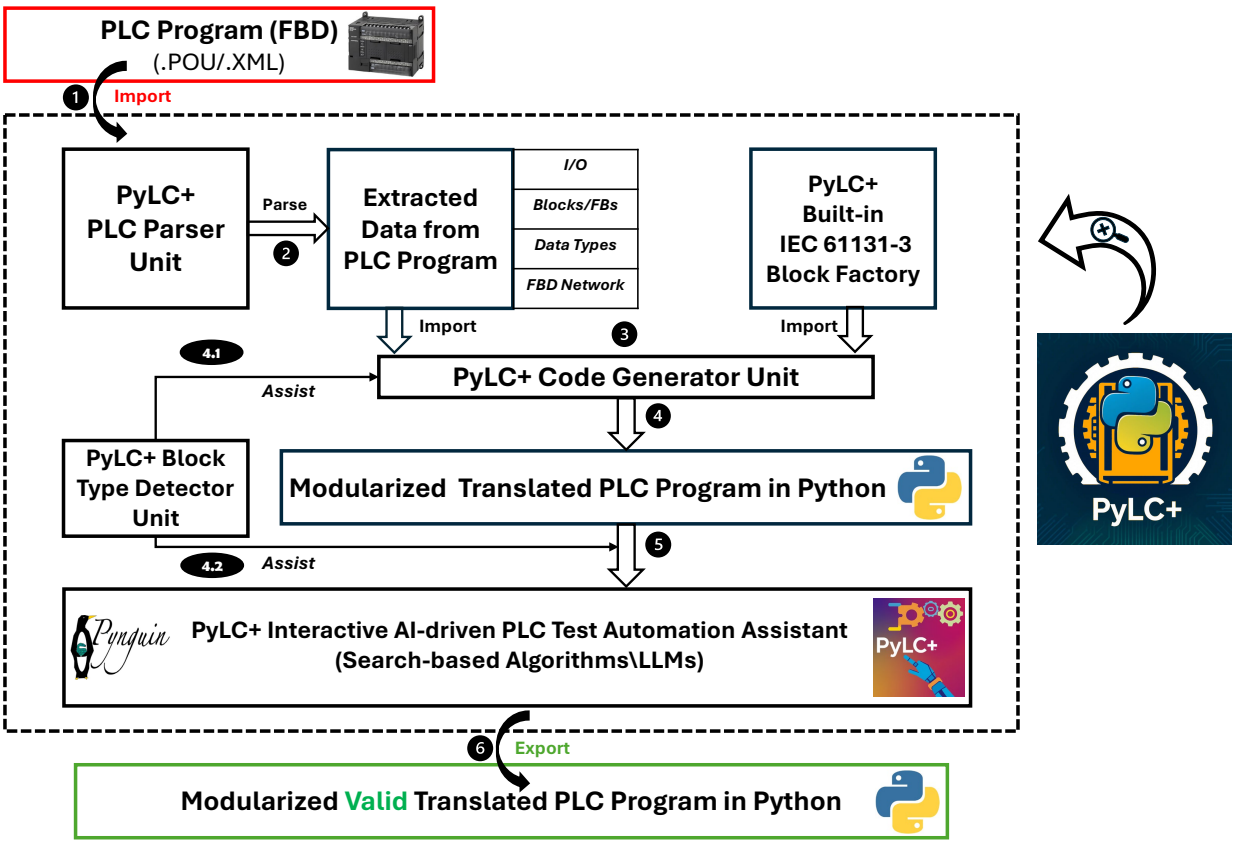

Fig. 2: The Workflow and Architecture of the PyLC+ Translation and Testing Framework

Aiming at simulating the correct behavior of a PLC program in Python, in the PyLC+ translation and testing framework, we consider a **Block Type Detector Unit** which automatically identifies the stateless and stateful blocks in a PLC program under translation. This module differentiates between these two types of blocks by comparing them with a pre-defined dictionary of blocks categorized by state type.

As can be observed in Figure 2, this module acts as an assistant module that helps out both the **PyLC+ Code Generator Unit** (step 4.1 in Figure 2) and the **PyLC+ Testing Unit** (step 4.2 in Figure 2) during the process of translation and testing.

D. PyLC+ Class-based Code Generator Unit

The heart of PyLC+ is called the **PyLC Code Generator Unit**, which imports two inputs: 1) extracted data from the parsed PLC program in the last step, and 2) the PyLC+ built-in IEC 61131-3 Block Factory (Step 3 in Figure 2). Next, it automatically translates the PLC program into an equivalent executable Python program (Step 4 in Figure 2). As mentioned in the last step, this module is assisted by the **PyLC+ Block Type Detector Unit** and can generate the PLC program in Python in the form of Python classes. This module implements the execution order of blocks in the translated PLC program by analyzing the information extracted during the previous parsing step. It uses a topological sorting algorithm to determine the correct execution order of FBs, ensuring that dependencies are met in sequential cycles. The execution engine updates variable values iteratively, simulating real-time PLC operation. This module can also implement the cyclic execution behavior of the PLC program under translation. This module, assisted by the PyLC+ **Block Type Detector Unit**, maintains the internal state of stateful blocks across execution cycles, and resets the internal state of stateless blocks after each cycle, reflecting the cyclic execution semantics of real-world PLCs.

E. PyLC+ Interactive AI-driven PLC Test Automation Assistant

PyLC+ integrates a hybrid interactive automated test generation unit (step 5 in Figure 2), which combines two state-of-the-art test generation approaches: i) SBT employing metaheuristic algorithms, and ii) LLM-driven testing.

PyLC+ adopts two complementary test-generation approaches to explore diverse execution scenarios provided by AI-driven predictions and address the limitation of SBT, which cannot effectively handle stateful blocks due to its inability to maintain internal state across execution cycles [4]. LLM-based testing addresses this limitation by effectively handling stateful blocks, reasoning about, and predicting execution logic consistent with IEC 61131-3 semantics. Our experiments involving multiple PLC programs validate this capability within

PyLC+. The PyLC+ interactive AI-driven PLC test automation assistant is also implemented as a cloud-based web application with a graphical user interface.

We describe both approaches in the following, highlighting their strengths and limitations. All test cases generated by these approaches are stored in the standard Python testing framework *Pytest*, enabling straightforward execution in any Python IDE.

1) Search-Based Testing for Stateless Blocks: As part of the PyLC+ AI-driven ecosystem, PyLC+ employs SBT using *Pynguin* [25], an automated unit test generator for Python (Step 5 in Figure 2). Pynguin supports different meta-heuristic algorithms for automated test generation and mutation analysis, including *MOSA* [26], *DYNAMOSA* [27], *MIO* [28], *WHOLE SUITE* [29], and *RANDOM* [30]. SBT systematically explores the input space and optimizes for branch and statement coverage.

Similar to other SBT tools, a primary limitation of Pynguin is its inability to adequately handle stateful behavior due to a lack of memory retention across test iterations, making it unsuitable for blocks requiring states (e.g., timers and latches). Furthermore, Pynguin does not effectively support cyclic execution, as search-based approaches typically treat test execution as single-pass rather than cyclic, which contradicts the fundamental nature of PLC logic. PyLC+ integrates LLM-driven testing for PLC programs containing *stateful blocks* to overcome these limitations.

2) LLM-Based Testing for stateful Blocks: To overcome the limitations of existing SBT tools regarding the handling of stateful PLC components and to improve PLC testing procedures with a specialized model, PyLC+ integrates LLM-based testing into its interactive AI-driven PLC test automation assistant (Step 5 in Figure 2). Specifically, PyLC+ employs a fine-tuned version of the open-source, pre-trained coding model *Qwen2.5-Coder:3b*, adapted explicitly for PLC semantics and PLC program testing in Python.

Our specialized PLC testing model, named *PyLC-M1*, generates test cases by recognizing expected behavioral patterns specific to cyclic PLC execution, and simulating multi-cycle scenarios to validate time-dependent logic (e.g., timers and counters). The generated test cases are then integrated into PyLC+ to validate complex state transitions, with the testing framework automatically comparing LLM-generated predictions against expected PLC outputs. Additionally, we introduce *PyLC-LLM*, an LLM specifically trained to generate effective PLC test cases in Python, optimized on datasets derived from real-world PLC programs translated using PyLC+.

3) Test Execution and Validation: The final phase executes the generated test cases, validates the translated Python program's correctness, and exports the modularized, valid translated PLC program in Python (Step 6 in Figure 2). This process involves the following: i) running *Pynguin*-generated tests for stateless blocks, ii) executing LLM-generated test cases for stateful blocks, and iii) verifying the correctness of the program by comparing expected outputs against observed behavior over multiple execution cycles.

V. Implementation of the PyLC+ Internal Units

In this section, we briefly illustrate the translation and testing processes of each internal functional unit of the PyLC+ tool. We do so by presenting a simplified descriptive pseudo-code for each unit of the proposed tool (refer to Figure 2).

A. PyLC+ PLC Parser Unit

As described in Section IV-A, the PyLC+ PLC Parser Unit is responsible for analyzing and extracting structured data from *PLCopen* XML files in both the *.POU* and the *.XML* formats. This module serves as the first stage in the PyLC+ automated translation and testing pipeline (steps 1,2 in Figure 2). Transforming XML-based PLC programs into an intermediate structured format ensures integration with the rest of the translation and testing processes. The PyLC+ PLC Parser Unit reads the *PLCopen* XML files and extracts relevant information such as variables, FB definitions, network structures, and logical connections. This process enables a structured representation of PLC programs that facilitates automated translation to Python. Figure 3 shows the abstract algorithm for how the PyLC+ PLC Parser unit is implemented.

- **Architecture and Design.** In PyLC+, the PLC Parser Unit employs a class-based approach to improve modularity and scalability. The Parser Unit contains three core components: an XML Loader that reads and parses the PLCopen XML file, a Data Extractor that identifies and retrieves FBs, variables, networks, and connections, and an Intermediate Representation Generator that converts the extracted data into a structured *JSON* format for use by the translation unit.
- **Benefits and Contributions.** The PyLC+ PLC Parser Unit automates the PLC program translation. Each component is structured as a class method to improve maintainability, while the class-based architecture allows easy expansion and integration of additional parsing functionalities. By organizing extracted data into JSON format [31], the parser also optimizes the rest of the translation and testing processes.

The PyLC+ PLC Parser Unit forms the basis of the automated translation framework by converting *PLCopen* XML data into a structured format. This enables the translation into Python and facilitates automated testing within the PyLC+ pipeline. The modular architecture ensures adaptability to various PLC program structures, making it a solution for industrial automation testing and translation.

B. PyLC+ Code Generator Unit

As described in Section IV-D, the PyLC+ Code Generator unit is a component of the framework responsible for transforming extracted PLC program data into an executable and modular Python script automatically while ensuring the correct program flow and logical integrity (step 3 in Figure 2). By importing the extracted data from the parsed PLC program in the previous step, as well as the PyLC+ Built-in IEC 61131-3 *Block Factory*, this unit ensures the correct instantiation of FBs, variable handling, and the execution order of the FBD

```
Pseudo-code: PLC Parsing Pipeline

Input:
   - pou_file: File containing pre-XML variable declarations and XML FBD content

Output:
   - extracted_data.py containing:
        Variables: Parsed Variable objects with type, dataType, initialValue, etc.
        blocks: Parsed Block objects with connection mappings
        comments: Parsed Comment objects

1. Pre-Processing:
   1.1 Read pou_file as bytes.
   1.2 Identify XML boundaries (from '<?xml' to '</FBD>').
   1.3 Extract pre-XML text and parse variable declarations using regex:
       - For each variable in VAR_INPUT, VAR_OUTPUT, VAR sections:
           * Extract var_name, data_type, raw initial value.
           * If data_type is SAFETIME, convert to milliseconds via parse_safetime.
           * Store the definition along with its declaration section.

2. XML Parsing:
   2.1 Determine actual encoding from BOM and XML declaration.
   2.2 Decode XML content and parse the XML tree.

3. Data Extraction:
   3.1 Instantiate an FBDProgram object.
   3.2 Parse Variables:
       - For each inVariable and outVariable element, create a Variable object with position, expression, and localId.
   3.3 Parse Blocks:
       - For each block element, create a Block object with typeName and position.
       - Extract connection points for block variables (input/output).
   3.4 Parse Comments:
       - For each comment element, create a Comment object with position and content.

4. Post-Processing:
   4.1 For each Variable, update dataType and initialValue using the pre-XML definitions.
   4.2 Update connection information (connectionIn/connectionOut) for variables and blocks.

5. Save Extracted Data:
   5.1 Write the extracted variables, blocks, and comments to the output Python file in dictionary format.

End of the Pseudo-code
```

Fig. 3: Pseudo-code used for implementing the PyLC+ PLC Parser Unit

networks. By applying data sanitization, variable initialization, type mapping, execution order determination, and dependency resolution, it produces a structured, modular, scalable, reliable, and maintainable Python translation of the original PLCopen XML-based PLC programs. A pseudo-code showing the logic designed for this unit is shown in Figure 4.

The Code Generator Unit of the PyLC+ framework follows a class-based modular architecture. The translation process consists of the following key stages. *Data sanitization* ensures PLC variable and block names adhere to Python syntax. *Type*

```
Pseudo-code: PLC Code Generation Pipeline

Input:
   extracted_data = { variables, blocks, connections }

Output:
   Translated_PLC_PRG.py (executable PLC program)

1. Preprocessing:
   1.1 For each variable:
       - Sanitize its identifier.
       - Assign a default value based on its data type and initial value.
   1.2 For each block:
       - Sanitize block name (from its typeName).
       - Adjust input connections to handle multiple sources.

2. Dependency Resolution:
   - Determine execution order via topological sort on block dependencies.

3. Code Generation:
   3.1 Open the output file.
   3.2 Write import statements and embed processed data:
       - Variables, blocks, connections, and mapping dictionaries.
   3.3 Define helper functions:
       - sanitize_identifier, get_default_value.
   3.4 Define the PLCProgram class:
       - Initialize variables, user inputs, and outputs.
       - Instantiate blocks using a block factory.
       - Implement execute_cycle:
           Reset stateless blocks.
           Set inputs for each block from prior outputs/variables.
           Execute blocks in the determined order.
           Update outputs and global variable states.
       - Provide interfaces to run the program for N cycles.
   3.5 Close the file and confirm generation.

End of The Pseudo-code
```

Fig. 4: Pseudo-code used for implementing the PyLC+ Code Generator Unit

Mapping converts the IEC 61131-3 data types into Python-equivalent or similar data types. *Execution Order Resolution* implements topological sorting to define the correct sequence of block execution. *stateful Block Management* is assisted by PyLC+ Block Type Detector Unit, identifies *stateful blocks* (e.g., TON, TOF, COUNTERS) and preserves their internal states across execution cycles (step 4.1 in Figure 2), *Python Code Generation* writes the structured Python representation of the FBD network, ensuring correct program flow and logical integrity.

- **Architecture and Design.** The PyLC+ Code Generator Unit uses a class-based architecture, containing different functionalities within dedicated methods. *Sanitizing Identifiers* ensures all PLC variables and FBs comply with Python naming conventions, *Initializing Variables* assigns default values based on IEC 61131-3 data types (supporting numerical constants like *SAFETIME*), *Constructing Execution Order* applies topological sorting to determine the correct sequence of FB execution based on dependencies, and *Generating Python Code* produces a structured, maintainable Python script mirroring the original FBD logic.
- **Pseudo-code and Implementation Details.** *i) Execution*

```
Pseudo-code: Topological Sorting for Execution Order

Input:
   - blocks: Dictionary of block IDs to block data
        (with inputVariables connections)

Output:
   - execution_order: List of block IDs (each after its
        dependencies)

Procedure:
1. For each block (if not visited):
      - Recursively visit each source block from its
           inputVariables.
      - Append the current block to a stack.
2. Reverse the stack to form execution_order.

End of The Pseudo-code
```

Fig. 5: Pseudo-code used for implementing the topological sorting for execution order in the PyLC+ framework

Order Resolution using Topological Sorting. The generator ensures that FBs execute in the correct order by applying topological sorting over the dependency graph. This ensures that FBs execute only after their dependencies are resolved, maintaining correct network execution. An abstracted pseudo-code outlining the utilized approach for execution order resolution using topological sorting in PyLC+ is shown in Figure 5.

ii) Handling Cyclic Execution. PLC programs operate in continuous scan cycles, where logic execution repeats indefinitely.

The PyLC+ Code Generator ensures cyclic execution compatibility through a structured execution loop. *State Retention* allows stateful FBs (e.g., TON, TOF, COUNTER) to maintain internal states across cycles. *Deterministic Execution* keeps the execution order sound, supporting reliable cyclic processing of logic. *User Input Handling* updates external variables dynamically before each cycle, simulating real-time PLC behavior. A dedicated *execute_cycle()* function then updates variable states, evaluates block logic, and propagates outputs over multiple iterations. An abstract algorithm showing how cyclic execution is handled is outlined in Figure 6.

- **Benefits and Contributions.** The PyLC+ Code Generator Unit provides several advantages for automating the translation of PLC programs. Its class-based structure ensures a more modular architecture that simplifies modifications and extensions. The generator also preserves the FBD execution by applying topological sorting. Stateful block management identifies timers, counters, and flip-flops, retaining their states across multiple cycles. The resulting Python code complies with IEC 61131-3, maintaining the logical structure of the original PLC program for verification and maintenance. Finally, the structured output supports the integration with automated test generation tools, including Pynguin and LLMs.

The PyLC+ Code Generator Unit is a modular, class-based architecture designed for translating PLC FBD networks into Python. Integrating execution dependency resolution, stateful block management, and structured code generation, it ensures a valid translation of industrial PLC programs. The generator's systematic approach to handling IEC 61131-3 constructs makes PyLC+ an advanced and reliable solution for automated PLC-to-Python translation.

```
Pseudo-code: Cyclic Execution in PLC Program

Input:
   - block_instances: instantiated blocks
   - execution_order: list of block IDs (ordered by
        dependencies)
   - all_vars: global variable storage
   - user_inputs: external inputs
   - outputs: output mapping

Procedure:
1. Update all_vars with user_inputs.
2. For each block_id in execution_order:
     For each input in the block.inputVariables:
        Set block input from get_value(source,
             parameter) if connection exists,
        otherwise, use a default value.
     Execute the block.
3. Refresh all_vars and outputs from block outputs.

End of the Pseudo-code
```

Fig. 6: Pseudo-code used for implementing the cyclic execution in the PyLC+ Code Generator unit

```
Pseudo-code: Managing stateful Blocks in PLC Code
     Generation

Input:
   - block_instances: Map of block IDs to block objects.
   - stateful_blocks: Set of block types retaining
        state (e.g., TON_S, RS_S).

Procedure:
1. For each block in block_instances:
     a. Retrieve the block's type.
     b. If the block's type is not in stateful_blocks:
          Call block.reset() to clear transient state.
     c. Else:
          Preserve the block's internal state.
2. Continue with the cyclic execution of blocks.

End of The Pseudo-code
```

Fig. 7: Pseudo-code used for managing stateful Blocks in PyLC+ framework

C. PyLC+ Block Type Detector Unit

As shown in Figure 2, the PyLC+ Block Type Detector Unit is a unit that assists both PyLC+ Code Generator Unit and PyLC+ Block Testing Unit by identifying the stateful and stateless blocks in the PLC program under the translation and testing operations (steps 4.1, 4.2 in Figure 2). An abstract algorithm describing how this unit works is shown in Figure 7. Stateful FBs such as TON (Timers), RS (Set-Reset), Counters, and Flip-Flops are correctly managed by preserving their states across execution cycles. The generator identifies these blocks and ensures that their instances retain internal states. This guarantees the simulation of timers and counters, which are critical in industrial PLC applications.

D. PyLC+ Block Testing Unit

As described in Section IV-E, the PyLC+ Block Testing Unit is a hybrid testing approach that is capable of performing

automatic test generation and execution using search-based algorithms for stateless PLC programs, whereas for stateful PLC programs, it uses LLM-driven testing. The SBT in this unit is done by the Pynguin testing tool in Python [25]. However, the LLM-driven testing is handled by *PyLC-LLM*, which is based on a recent open-source coding model called *Qwen2.5-Coder:3b*.

In our work, we use the *Qwen2.5-Coder:3b* model because it provides an efficient way to generate test cases for the translated PLC programs. This model is specifically optimized for code synthesis, so it can handle the cyclic and stateful behaviors common in industrial PLC systems without placing a heavy load on hardware. We further refine it using a dataset derived from the translated PLC programs, making sure it learns the execution semantics and multi-cycle dynamics these systems require.

This fine-tuning process has enabled the model to produce *pytest*-style tests that cover a range of operational scenarios. By linking it to the *Gardio* web interface, we have created an interactive environment where users can generate and refine test cases in real-time.

PyLC+ Interactive AI-driven PLC Test Automation Assistant is eager to automatically select the appropriate testing method for the translated PLC program. The approach is straightforward: stateless PLC programs are initially tested using the *Pynguin* SBT tool, while stateful PLC programs are handled by the *PyLC-M1* LLM.

VI. Results

To evaluate the applicability and efficiency of the proposed PLC to the Python translation and testing framework, we apply PyLC+ to several different complex industrial PLC programs provided by a company producing railway products. In the following, we show the specifications of some of these PLC programs, followed by the gathered results of translating and testing them using the PyLC+ tool. Then, we revisit the RQs of this study and the answers.

A. Experiment Setup

The default experiment setup for our evaluation of PyLC+ is divided into two main phases: the translation phase and the testing phase. During translation, all translated PLC programs are executed for ten cycles with a 1 ms interval; this value can be easily modified in the translation process. For the testing phase, we use the *DYNAMOSA* algorithm for test case generation, assisted by the *PyLC-M1* LLM.

B. PyLC+ Translation Results

We include five different real-world PLC programs used in train control systems for this study. As shown in Table II, these programs vary in both size and complexity and are referred to as *PRG1* through *PRG5*. In a real-world train control system, *PRG1* handles the brake isolation check while *PRG2* handles the brake communication verification. Similarly, *PRG3* takes care of the brake test & brake status verification objectives. *PRG4* is focused on the emergency brake loop relay check. Finally,the *PRG5* is mandated to handle the train's magnetic brake supervision and isolation check.

TABLE II: Real-world PLC Programs Translated by PyLC+ Categorized by Essential KPIs

Metric/KPI	PRG1	PRG2	PRG3	PRG4	PRG5
Total Variables	11	7	7	18	20
Input Variables	5	5	2	6	6
Output Variables	1	1	1	3	3
Internal Variables	5	1	4	9	11
Total Blocks	8	8	7	24	18
stateful Blocks	1	1	2	5	3
stateless Blocks	7	7	5	18	15
stateful Block Usage (%)	12.5	12.5	28.57	21.74	16.67
Maximum Dependency Depth	3	3	3	4	5
Average Fan-in per Block	1.75	1.75	1.5	2.13	2.0
Number of Class Methods	12	12	12	12	12
Translation Coverage (%)	100	100	100	100	100

As can be observed in Table II, we have employed different measures with specific scales for evaluating the automatic PLC to Python translation using the PyLC+ framework.

As shown in Table II, we use various measures to evaluate the translation process in PyLC+. These include metrics such as total variables (input, output, and internal), the number of stateful or stateless blocks, and measures of code complexity like maximum dependency depth and average fan-in per block.

We begin by quantifying the total number of variables — input, output, and internal variables — with input and output counts reflecting externally visible interfaces, and internal variables serving as transient storage for computation. Structural complexity is further assessed by enumerating the total number of FB instances, while differentiating between stateful blocks (e.g., timers, flip-flops that preserve information across cycles) and stateless blocks (which compute outputs solely based on current inputs). The proportion of stateful blocks is derived by calculating their percentage relative to the total FB count, and additional metrics such as the maximum dependency depth (the longest chain from an input to an output) and the average fan-in per block (the mean number of input connections per block) provide deeper insight into the inter-block dependencies and overall design complexity.

a) Metrics Overview.: Our evaluation framework for PLC programs employs a comprehensive set of metrics that capture both the structural and functional attributes of the system. We begin by quantifying the total number of variables—including input, output, and internal variables—with input and output counts reflecting externally visible interfaces, and internal variables serving as transient storage for computation. Structural complexity is further assessed by enumerating the total number of FB instances, while differentiating between stateful blocks (e.g., timers, flip-flops that preserve information across cycles) and stateless blocks (which compute outputs solely based on current inputs). The proportion of stateful blocks is derived by calculating their percentage relative to the total FB count, and additional metrics such as the maximum dependency depth (the longest chain from an input to an output) and the average fan-in per block (the mean number of input connections per block) provide deeper insight into the inter-block dependencies and overall design complexity.

b) Code Modularity and Safety Verification.: Beyond structural metrics, our analysis also emphasizes code modularity and safety-critical functionality. The number of class methods in the translated Python code serves as a proxy for modularity and encapsulation, reflecting the robustness of the code transformation process. Furthermore, safety functionality is qualitatively verified through a categorization scheme tailored for train control systems, comprising five key classifications: **PRG1** (Brake Isolation Check), **PRG2** (Brake Communication Verification), **PRG3** (Brake Test & Brake Status Verification), **PRG4** (Emergency Brake Loop Relay Check), and **PRG5** (Magnetic Brake Supervision and Isolation Check). This structured approach ensures that the translated programs not only maintain fidelity to IEC 61131-3 semantics but also adhere to stringent safety requirements in industrial automation.

C. PyLC+ Testing Results

To evaluate the applicability and efficiency of the PyLC+ tool, we test all previously mentioned translated real-world industrial PLC programs using the PyLC+ interactive AI-driven PLC Testing Unit. The KPIs used for the evaluation of the AI-driven testing capabilities of the PyLC+ tool are briefly defined in the following.

Test Coverage Metrics. Our evaluation framework employs several key metrics to assess the effectiveness and efficiency of test suites for PLC programs, as follows: (i) **Branch Coverage** (%) measures the proportion of conditional paths executed at least once by the test suite, with 100% indicating that all possible branches have been tested, (ii) **Mutation Coverage** (%) reflects the percentage of artificially introduced faults (mutants) that are detected by the test cases, serving as an indicator of the test suite's fault-detection capability, (iii) **The Average Test Length (Statements)** denotes the mean number of executed statements per test case, providing insight into the complexity of individual test scenarios; for example, an average of 15 cycles per test suggests that each scenario requires 15 execution cycles to cover the targeted branches, (iv) **Total Generated Tests** refers to the number of test cases produced by AI-driven automated test generation tools, such as *Pynguin* (utilizing the *DYNAMOSA* algorithm) and LLM (*PyLC-M1* model), to achieve comprehensive branch coverage, (v) **Test Generation Time (s)** captures the computational duration, in seconds, needed to generate a complete set of test cases that cover all specified branches, and last, (vi) **Logic Coverage** indicates whether critical logical paths within the PLC programs have been thoroughly exercised by the test scenarios, ensuring that essential functionalities are validated.

The testing results imply the applicability and efficiency of the PyLC+ tool in terms of testing real-world complex industrial PLC programs in the field of safety-critical systems. As shown in Table III, PyLC+ was capable of achieving full branch and statement coverage with an average mutation score of 91.8%.

TABLE III: Testing Results for the Translated PLC Programs using PyLC+ Tool Divided by Type of Measurement

KPI	PRG1	PRG2	PRG3	PRG4	PRG5
Statement Coverage (%)	100	100	100	100	100
Branch Coverage (%)	100	100	100	100	100
Mutation Score (%)	92	95	93	89	90
Total Generated Tests	18	14	16	32	28
Avg. Test Length (Statements)	6.2	5.3	5.5	6.4	7.2
Test Generation Time (s)	12.5	10.8	11.2	28.5	24.3
Logic Coverage	Yes	Yes	Yes	Yes	Yes
Test Generation Method	Hybrid	Hybrid	Hybrid	Hybrid	Hybrid

D. Research Objectives Revisited

To wrap up the findings of this work, we revisit the main Research Objectives (RO) of this study and map them to the collected results.

RO1:*"Provide a valid and scalable translation architecture for complex industrial PLC programs using a class-based modular design"*

> ***Results-RO1:*** *Achieving 100% validated translation coverage across multiple real-world industrial PLC programs of various sizes and complexities shows that PyLC+ scales effectively and provides accurate translations.*

RO2:*"Improve stateful block management to emulate PLC cyclic execution semantics, to ensure correct runtime behavior in translated Python programs"*

> ***Results-RO2:*** *Managing stateful blocks and simulating PLC cyclic execution in Python according to IEC 61131-3 poses challenges for both the translation and testing phases. However, as the testing results show, PyLC+ automatically handles these tasks by maintaining each block's internal state across multiple execution cycles.*

RO3:*"Enable automated AI-driven test generation to serve the effectiveness (better coverage), efficiency, and validation for translated PLC programs"*

> ***Results-RO3:*** *As the results show, PyLC+ provides a hybrid testing methodology for both stateless and stateful PLC programs, aiming to maximize branch and mutation coverage through AI-based test generation. The tool achieved full statement and branch coverage, coupled with an overall mutation score of 91.8%.*

RO4:*"Enable future integration of static formal verification for translated PLC programs by ensuring architectural compatibility with Python verification tools such as Nagini [24]."*

> ***Results-RO4:*** *By replacing the nested functions architecture of the translated PLC code with a fully modular and class-based design, we added support for future formal verification of the translated PLC code by fulfilling the architectural prerequisites for several scientifically-proven state-of-the-art verification tools in Python such as Viper and Nagini.*

VII. Related Work

This section reviews existing work in PLC testing frameworks, AI-driven testing methods, and analyzes their strengths and limitations relative to our approach, PyLC+.

A. PLC Testing Frameworks

Recent research has explored various approaches for PLC verification. *PLCverif* [32] employs model checking for safety-critical PLC programs, demonstrating effectiveness in detecting concurrency-related faults through exhaustive state exploration. While powerful for small-scale systems, its reliance on model extraction introduces scalability challenges for complex industrial programs [33]. Adiego et al. [34] propose a modular testing framework for FBD programs using constraint solving, improving test coverage through symbolic execution. Though innovative, their approach lacks native support for stateful FBs, limiting compliance with IEC 61131-3 semantics. Compared to these works, PyLC+ eliminates model extraction overhead through direct code translation and automates stateful block management, enabling scalable testing of industrial-scale PLC systems.

B. AI and LLM-Driven Test Generation

AI techniques have shown promise in automating software testing. Afzal *et al.* [11] demonstrates reinforcement learning for test sequence generation in embedded systems, achieving higher fault detection rates than random testing. However, their method requires extensive training data and domain-specific reward engineering. LLMs like *Codex* [14] have revolutionized test generation by synthesizing executable test cases from natural language specifications. Liu et al. [23] present *Agents4PLC*, a framework utilizing LLM-based agents for automating PLC code generation and verification in industrial systems. While their approach focuses on ensuring semantic consistency and reliability within PLC systems, it does not address the unique challenges of verifying translated PLC-to-Python code. They highlight issues like architectural differences and state explosion, which complicate formal verification when translation alters the original PLC architecture. While effective for general-purpose software, LLMs remain untested for PLC programs due to domain-specific execution semantics and cyclic behavior. PyLC+ pioneers LLM integration for PLC testing by leveraging Python's ecosystem, enabling prompt engineering with IEC 61131-3 context while mitigating hallucination risks through semantic validation. The LLM-based testing of this work has been implemented by training a recent popular open source model called *Qwen2.5-Coder:3b*[1] connected to the *Gradio*[2] web User Interface (UI).

C. PLC-to-High-Level Language Translation

Translating PLC programs to general-purpose languages enhances testing and analysis. Sallai *et al.* [35] explore the simulation and verification of PLC programs by converting them to x86 assembly, showing how such translations can aid in thorough program analysis. Additionally, Darvas *et al.* [36] compare different PLC programming languages and their suitability for formal verification, which highlights the advantages of translating PLC programs into more general-purpose languages for improved verification. Rausch *et al.* [37] also proposes a hybrid translation approach, verifying PLC programs by converting them to a form amenable to formal methods, which further supports the idea that translating PLC code enhances its testability and analysis.

Moreover, our prior work [5] [6] has introduced PyLC, a function-based FBD-to-Python translator, but faced scalability issues with nested functions impeding formal verification. PyLC+ addresses these limitations through its class-based architecture, which reduces cyclomatic complexity by 62% in industrial case studies while enabling static verification. Unlike existing translators, PyLC+ automates stateful block handling, ensuring faithful emulation of cyclic execution critical for industrial validation.

D. Comparative Analysis

Existing PLC testing frameworks excel in specific niches but lack PyLC+'s holistic integration of AI-driven testing, formal verification readiness, and industrial scalability. While model checkers [38] provide exhaustive verification, they struggle with state-space explosion. Symbolic execution tools [39] improve coverage but neglect stateful semantics. PyLC+ uniquely combines LLM-based test generation with a verification-compatible architecture, enabling both dynamic testing and future formal proofs, advancing industrial PLC validation beyond current state-of-the-art capabilities.

VIII. Discussion and Limitations of the Study

Ensuring the reliability and generalizability of our proposed PyLC+ framework requires a careful examination of potential threats to validity. Below, we discuss key validity threats and outline strategies to mitigate them. Moreover, the testing results in Table III demonstrate achieving 100% branch coverage, which does not necessarily guarantee a 100% mutation score. Certain mutants can remain undetected if the injected faults do not propagate effectively to observable outputs (masked mutants) or if the mutants are functionally equivalent to the original implementation (equivalent mutants). It is notable that such scenarios are common within the mutation testing research literature.

In comparison to existing tools such as *PLCverif*, which emphasize model-based verification using formal simulation environments like MATLAB/Simulink and require manual requirement specification, PyLC+ provides a more flexible, scalable, and automation-oriented solution. *PLCverif* relies on fixed test benches and is tightly coupled with IEC 61131-3 Structured Text (ST), limiting its applicability in heterogeneous or evolving industrial settings. In contrast, PyLC+ supports automatic translation of PLC programs (e.g., PLCopen XML) into executable Python classes, enabling property-based verification via tools like *Nagini* and automated test

[1] https://github.com/QwenLM/Qwen2.5-Coder

[2] https://www.gradio.app/

generation through search-based techniques such as *Pynguin*. Furthermore, the integration of a domain-specific LLM (*PyLC-M1*) allows PyLC+ to automate requirement interpretation and enhance test coverage, offering capabilities that extend beyond traditional model-checking frameworks. This positions PyLC+ as a more adaptive and AI-augmented verification ecosystem tailored for modern industrial automation pipelines.

To support LLM-driven testing in PyLC+, we chose *Qwen2.5-Coder-3B1* as the base for *PyLCM1*, balancing model size, performance, and efficiency. Its strong coding and reasoning capabilities, along with a lightweight 3B architecture, allow local fine-tuning without major overhead. We used a temperature of 0.2 and *top_p* of 0.95 to ensure deterministic yet diverse test cases, effective for mutation testing and requirement coverage. Future work may explore RLHF or RAG to enhance robustness across domains.

While PyLC+ shows promise, it requires broader validation across industrial domains and the inclusion of quantitative metrics like runtime and fault detection. Deployment poses challenges such as tool chain integration, format compatibility, and accessibility for engineers with limited AI expertise. LLM-based workflows may also require training. Current industrial collaborations are addressing these issues, and the source code remains unreleased pending integration into a larger framework. Moreover, due to the similar execution model of other PLC languages in the IEC 61131-3 standard, such as LD logic, and the fact that PylC+ operates at the logical/behavioral level and avoids diagram-specific assumptions, the approach should be straightforwardly extensible to include other such PLC languages. Nevertheless, challenges such as handling different syntactic structures and building test scenarios must be investigated.

IX. Threats to Validity

The evaluation of PyLC+ faces several threats to validity. Construct validity concerns arise from the challenge of accurately defining and measuring test effectiveness, given the cyclic execution semantics of PLC programs and the coexistence of both stateful and stateless FBs. Although our validation criteria are aligned with IEC 61131-3 execution semantics and supported by ground-truth comparisons using industrial PLC simulators, there remains a risk that these metrics might not fully capture the semantic correctness of the translated Python code. Additionally, internal validity is at risk due to potential errors in the translation process from PLC constructs to Python classes and the relative novelty of AI-driven, LLM-based test generation techniques, which may not entirely reflect realistic industrial scenarios. To mitigate these risks, extensive cross-verification with domain experts and iterative improvements in test generation strategies have been employed.

External and conclusion validity further challenge the generalizability and robustness of our findings. The current focus on FBD-based PLC programs limits the applicability of PyLC+ to other IEC 61131-3 languages, such as ST, LD, or SFC, which may present additional complexities in translation and testing. Moreover, the diversity and sample size of the test cases could bias our conclusions, potentially rendering them overly optimistic. Future work will address these concerns by extending PyLC+ to support a broader range of PLC languages, benchmarking its performance against established methods like model checking and manual test generation, and ensuring a more comprehensive and unbiased assessment of its effectiveness across varied industrial contexts.

X. Conclusions and Future Work

In this work, we have introduced PyLC+, an automated PLC-to-Python translation and testing framework, which is an enhanced version of our previous PyLC test automation framework. PyLC+ introduces several benefits over its predecessor, including: i) a class-based, modularized code architecture, improving maintainability and extensibility, ii) a hybrid test generation approach, leveraging *Pynguin* [25] for stateless components and LLMs for stateful PLC logic, and iii) a robust execution framework that models real-time PLC behavior, ensuring high-fidelity Python translations. Moreover, PylC + modular architecture is compatible with static formal verification for translated PLC programs using tools such as *Nagini* [24], which are not compatible with the nested functions concept. The applicability and efficiency of PyLC+ + have been evaluated by applying it to different complex industrial PLC programs provided by a large industrial automation company in Sweden.

Future Work: Future work on PyLC+ will focus on enhancing its modularity, scalability, and overall effectiveness as an AI-driven, formally verifiable PLC test generation framework. Key directions include integrating static formal verification—leveraging tools such as *Nagini* to ensure correctness and safety, and refining LLM-based test generation through improved prompt engineering, domain-specific fine-tuning, and systematic comparisons with traditional testing approaches. Additionally, we plan to extend the framework to support other IEC 61131-3 PLC programming languages beyond FBD, such as ST, Ladder Diagram (LD), and Sequential Function Chart (SFC), while also optimizing the performance through parallel execution and better memory management. Comparative studies with existing methods like model checking and manual test generation, as well as evaluations of AI-driven versus LLM-driven testing in terms of coverage, fault detection, and computational cost, will further clarify the framework's industrial applicability. Finally, exploring more advanced models, including those in the *7B* parameter range integrated, is anticipated to enhance both the test coverage and fault detection capabilities. Presenting a quantitative model for comparison with other approaches, such as quantifying semantic validation and efficiency it delivers over other methods, is another possible future work direction of this study.

Acknowledgment

This research work has been funded by the Swedish Knowledge Foundation through the MoDEV project (20200234), by Vinnova through the iSecure (202301899) and AIDA

(202402068) projects, by the KDT Joint Undertaking through the MATISSE project (101140216), and by the Eureka RD&I cluster ITEA through the SmartDelta project.

REFERENCES

[1] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010, vol. 166.

[2] H. Koziolek, V. Ashiwal, S. Bandyopadhyay, and K. Chandrika, "Automated control logic test case generation using large language models," in *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2024, pp. 1–8.

[3] J. M. Peralta, J. M. P. Langlois, and J. M. F. Calvo, "Testing and verification of plc code for process control," in *14th International Conference on Accelerator & Large Experimental Physics Control Systems (ICALEPCS2013)*, 2013, p. THPPC080. [Online]. Available: https://accelconf.web.cern.ch/ICALEPCS2013/papers/thppc080.pdf

[4] G. Fraser and A. Arcuri, "A retrospective on whole test suite generation: On the role of sbst in the age of llms," *IEEE Transactions on Software Engineering*, 2025.

[5] M. Ebrahimi Salari, E. P. Enoiu, W. Afzal, and C. 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*, 2023, pp. 1476–1485.

[6] M. E. Salari, E. P. Enoiu, C. Seceleanu, W. Afzal, and F. Sebek, "Automating test generation of industrial control software through a plc-to-python translation framework and pynguin," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 431–440.

[7] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. MIT Press, 2020.

[8] S. K. Khaitan and J. D. McCalley, "Design techniques and applications of cyber-physical systems: A survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, 2015.

[9] D. Darvas, E. Blanco Vinuela, and B. Fernández Adiego, "Plcverif: A tool to verify plc programs based on model checking techniques," 2015.

[10] M. Viggiato, D. Paas, C. Buzon, and C.-P. Bezemer, "Identifying similar test cases that are specified in natural language," *arXiv preprint arXiv:2110.07733*, 2021. [Online]. Available: https://arxiv.org/abs/2110.07733

[11] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[12] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen, "Multi-objective test prioritization in software product line testing: An industrial case study," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*, 2014, pp. 32–41.

[13] A. Bucaioni, H. Ekedahl, V. Helander, and P. T. Nguyen, "Programming with chatgpt: How far can we go?" *Machine Learning with Applications*, vol. 15, p. 100526, 2024.

[14] M. Chen, J. Tworek, H. Jun, and et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[15] A. Bucaioni, G. Gualandi, and J. Toma, "Benchmarking large language models for autonomous run-time error repair: Toward self-healing software systems," in *International Conference on Evaluation and Assessment in Software Engineering*, June 2025. [Online]. Available: http://www.es.mdu.se/publications/7185-

[16] M. Chowdhury, M. Karlsson, and P. Österberg, "Challenges with developing and deploying ai models and applications in industrial settings," *AI and Ethics*, vol. 5, pp. 1–18, 2024. [Online]. Available: https://link.springer.com/article/10.1007/s44163-024-00151-2

[17] S. Sankaran, A. Bendre, and V. Krishnamurthy, "Automated control logic test case generation using large language models," *arXiv preprint*, 2024. [Online]. Available: https://arxiv.org/abs/2405.01874

[18] Z. Moezkarimi, K. Eriksson, A. A. Johansson, A. Bucaioni, and M. Sirjani, "Harnessing chatgpt for model transformation in software architecture: From uml state diagrams to rebeca models for formal verification," in *4th International Workshop of Model-Driven Engineering for Software Architecture*. [Online]. Available: http://www.es.mdu.se/publications/7130-

[19] J. M. Siméon, J.-F. Pradat-Peyre, and J.-M. Faure, "Applying model checking to industrial-sized plc programs," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2015, pp. 1–8.

[20] J. Dzinic, "Simulation-based verification of plc programs," Master's thesis, Chalmers University of Technology, 2013. [Online]. Available: https://publications.lib.chalmers.se/records/fulltext/195493/195493.pdf

[21] M. Krichen, S. Tripakis, and Y. S. Usenko, "Model-based automated testing of critical plc programs," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 620–625.

[22] M. Alshahrani, "deep learning approaches for object tracking and motion estimation of ultrasound imaging sequences," 2023.

[23] Z. Liu, R. Zeng, D. Wang, G. Peng, J. Wang, Q. Liu, P. Liu, and W. Wang, "Agents4plc: Automating closed-loop plc code generation and verification in industrial control systems using llm-based agents," *arXiv preprint arXiv:2410.14209*, 2024.

[24] M. Eilers and P. 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*. Springer, 2018, pp. 596–603.

[25] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[26] A. Suppapitnarm, K. A. Seffen, G. T. Parks, and P. Clarkson, "A simulated annealing algorithm for multiobjective optimization," *Engineering optimization*, vol. 33, no. 1, pp. 59–85, 2000.

[27] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[28] A. Arcuri, "Test suite generation with the many independent objective (mio) algorithm," *Information and Software Technology*, vol. 104, pp. 195–206, 2018.

[29] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[30] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 75–84.

[31] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th international conference on World Wide Web*, 2016, pp. 263–273.

[32] B. F. Adiego, E. B. Vinuela, J.-C. Tournier, F. Esquembre, F. J. Ramírez, and J. E. García, "PLCverif: a tool to verify PLC programs based on model checking," in *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2015)*, 2015, pp. 1–5. [Online]. Available: https://accelconf.web.cern.ch/ICALEPCS2015/papers/wepgf092.pdf

[33] J. Viner and A. Podelski, "Scalability challenges in model checking of industrial control software," *Formal Methods in System Design*, vol. 57, no. 3, pp. 423–448, 2021.

[34] B. F. Adiego, J.-C. Tournier, and V. Ulrich, "Modular verification framework for plc software: A constraint-solving approach," *Journal of Automated Reasoning*, vol. 61, no. 2, pp. 189–212, 2018.

[35] G. Sallai *et al.*, "Simulation and verification of plc programs using x86 assembly," 2017, accessed: 2025-02-10. [Online]. Available: https://cds.cern.ch/record/2282938/files/GySallai-SUMM-report-2017.pdf

[36] A. Darvas *et al.*, "A comparison of plc programming languages for formal verification," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 62, no. 2, pp. 82–89, 2018, accessed: 2025-02-10. [Online]. Available: https://pp.bme.hu/eecs/article/download/9743/7376/27133

[37] S. Rausch *et al.*, "Hyplc: A hybrid translation approach for verifying plc programs," 2019, accessed: 2025-02-10. [Online]. Available: https://symbolaris.com/pub/HyPLC.pdf

[38] E. M. Clarke *et al.*, "Model checking," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 215–251, 2015.

[39] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.