A Verification-Aware Pipeline for Programmable Logic Controllers: From Function Block Diagrams to Verified Python Code

Mikael Ebrahimi Salari*, Eduard Paul Enoiu*, Cristina Seceleanu*, Marco Eilers†, Alessio Bucaioni*, Wasif Afzal*
*School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
Emails: {mikael.salari, eduard.paul.enoiu, cristina.seceleanu, wasif.afzal, alessio.bucaioni}@mdu.se
†Department of Computer Science, ETH Zurich, Zurich, Switzerland
Email: marco.eilers@inf.ethz.ch

***Abstract*—Translating programmable logic controller (PLC) programs into analyzable software artifacts is an important step toward enabling modern software testing and verification techniques in industrial settings. This paper presents a replication study in which we use PyLC+, a translation framework for IEC 61131-3 Function Block Diagrams (FBD), together with the Nagini verifier to perform functional correctness checks on industrial safety-critical logic. PyLC+ extracts block networks from PLCopen XML and generates executable Python models that preserve block semantics, signal flow, and scan-cycle behavior.**

We applied this workflow to a representative set of 13 industrial POUs from an electropneumatic brake control subsystem. All 13 POUs were verified: twelve using concrete or partially abstracted FBD models, and one using an abstract requirements-level specification whose 7-tuple specification initially exposed a bug in the Nagini verifier (a lack of support for tuples with more than six elements), which was fixed by the Nagini developers.

The results show that a combination of automated extraction, manual semantic reconstruction, and contract-based specification is sufficient to verify realistic industrial logic at the POU level. We also discuss the abstraction decisions, the Nagini bug uncovered by one POU, and the remaining missing block semantics that constrain how far automation can be pushed in future versions of PyLC+.

***Index Terms*—PLC, Formal Verification, PyLC+, Nagini, Viper, large language models, Industrial Automation**

I. Introduction

Programmable Logic Controllers (PLCs) continue to form the computational backbone of industrial automation systems. Their safety-critical role, particularly in transportation, energy, and process industries, has intensified the need for strong assurance techniques that go beyond testing on physical hardware. Formal verification is a natural candidate, yet its adoption in industry is limited by two long-standing obstacles: (i) the semantic gap between PLC languages (e.g., IEC 61131-3 Function Block Diagrams) and verification-oriented representations, and (ii) the substantial manual effort required to formulate specifications, harnesses, and environment assumptions.

Recent research has explored the idea of translating PLC programs into analyzable models in higher-level languages such as Python or C, thereby enabling the use of modern verification tools. However, these approaches are often demonstrated on simplified examples or depend on handcrafted models that do not reflect the complexity of industrial control logic. As a result, the question of practical feasibility, whether real PLC logic, with realistic block networks and safety patterns, can be meaningfully verified, remains insufficiently explored.

Our prior work, PyLC+ [1] translates an FBD program, an IEC 61131-3 graphical language where control logic is expressed using interconnected function blocks, stored in PLCopen XML format, a vendor-neutral representation of PLC programs defined by the PLCopen organisation, into executable Python code and automates test generation at scale. However, test adequacy is limited by coverage and the quality of the oracle. This work advances PyLC+ by integrating Nagini [2], a static verifier for Python that compiles annotated code to the Viper intermediate language and discharges verification conditions via Silicon and Carbon backend inside the Viper [3] verification infrastructure, to provide a high level of assurance of the obtained Python code. Our goal is not to showcase full automation, but to evaluate how far a semi-automated workflow can be pushed when applied to real industrial artifacts. PyLC+ extracts FBD networks from PLCopen XML, reconstructs block structures, preserves execution order, and generates executable Python models that encode combinational, stateful, and timing behavior. Verification harnesses, including functional contracts and domain assumptions, were created in a semi-automated manner: an LLM was used to generate scaffolding and assist with boilerplate, while semantic constraints, invariants, and pre-/postconditions were refined manually.

We apply this workflow to a set of 13 real-world industrial Program Organization Units (POUs) from a safety-critical electropneumatic brake control subsystem. These POUs cover a diverse set of control responsibilities, including redundant-sensor validation, pressure conversion, brake plausibility checking, magnetic brake supervision, and parking brake state evaluation. All 13 POUs were verified using Nagini's Silicon backend: twelve under concrete or partly abstracted FBD models, and one under an abstract requirements-level specification.

The latter POU initially exposed a bug in Nagini's handling of tuples larger than six elements, which the Nagini developers fixed based on our report, after which the harness verified successfully. The study highlights both the strengths and limits of translation-based verification: the approach succeeds for the majority of realistic control logic but still requires human judgment for semantic reconstruction and contract definition.

Our contributions are threefold. First, we demonstrate that translation-based modeling combined with manual contract design is sufficient to verify a substantial portion of a real industrial PLC subsystem. Second, we document the abstraction decisions, semantic gaps, and incomplete block-library coverage that prevented full verification of one POU. Third, we provide a detailed account of the workflow, including translation, manual refinement, and verification harness design, that future automated tools must support to make end-to-end PLC verification practical. The remainder of this work is structured as follows. Section II introduces the necessary background on PLC semantics, PyLC+, and the Nagini/Viper verification framework. Section III states the research goal and objectives. Section IV presents the system overview and verification-aware workflow. Section V details the harness architecture, PLC semantics, and verification scope used in the study. Section VI reports the verification results for 13 industrial POUs. Section VII evaluates the pipeline through structural and performance metrics. Section VIII discusses the findings and their limitations, and Section IX surveys related work. Section X concludes with directions for future research.

II. Background

PLCs are the backbone of modern industrial automation systems. They execute control logic in cyclic scan loops where inputs are read, the program is evaluated, and outputs are written. Their reliability and predictable timing make them central in manufacturing, transport, energy, and other safety-critical domains. The IEC 61131-3 standard defines several languages for programming PLCs, ranging from textual (Structured Text, Instruction List) to graphical (Ladder Diagrams and Function Block Diagrams).

FBD is a graphical, dataflow-oriented language in which engineers build control logic by connecting function blocks with wires. Each block represents a specific operation, such as Boolean logic (AND, OR), arithmetic (ADD, SUB), timers (TON, TOF), counters (CTU, CTD), or vendor-defined components. During each PLC scan cycle, data flows through these interconnected blocks, forming networks whose execution order is determined by the structure of the diagram. Combinational networks denote acyclic expressions; stateful elements (e.g., RS) and timers (e.g., TON) carry state across cycles. Understanding these blocks and how their state evolves across cycles is essential for translation, testing, and verification.

A small FBD example is shown in Figure 1. It illustrates basic logic blocks (AND, OR) and a stateful timer block (TON), which delays a rising input signal by a specified time.

Many industrial FBD programs rely on the IEC 61131-3 TON (on-delay) timer, whose behaviour depends on how the timer reacts to short input changes, often referred to as *glitches*. A timer maintains an internal state, the elapsed time (ET), which is carried from one scan cycle to the next. Two styles of timing are commonly observed in industrial code:

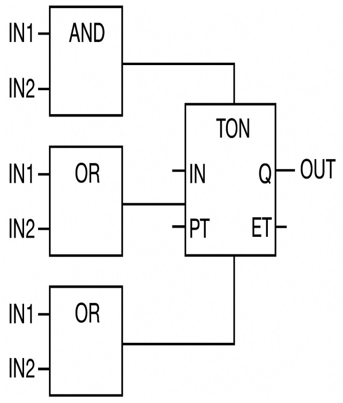

Fig. 1. A simple FBD example illustrating logic blocks and a TON timer.

- **Continuous-window semantics:** the input must remain TRUE for the entire preset window (PT). Any short FALSE glitch resets the timer back to zero. This is the behaviour of classical IEC-compliant TON blocks.
- **Accumulative semantics:** the timer accumulates time across cycles even if the input briefly drops to FALSE, as long as the total accumulated time reaches PT. This style appears in some vendor-specific libraries and legacy systems.

Example. Consider a preset time $\texttt{PT} = 5$ cycles and the input sequence

$$\texttt{IN} = [\texttt{TRUE, TRUE, FALSE, TRUE, TRUE, TRUE}].$$

Under *continuous-window* semantics, the FALSE in cycle 3 resets the timer, so the window of five consecutive TRUE cycles must start again at cycle 4. Counting cycles $4, 5, 6, 7, 8$ as the new window, the timer reaches its preset at the *end of cycle 8*. (Different informal explanations sometimes report this as "cycle 7" depending on whether the start of the window is counted inclusively or exclusively, so we adopt the cycle-end convention for clarity.)

Under *accumulative* semantics, the timer accumulates the two initial TRUE cycles (1–2) and the three post-glitch cycles (4–6), reaching a total of five, so the output Q becomes TRUE at the end of cycle 6.

These constructs appear frequently in industrial control applications and are referenced throughout the paper (e.g., TON, AND, OR, ADD). This short introduction sets the terminology used later in translation, testing, and verification discussions.

Nagini and Viper. *Nagini* [2] is a static verification tool for Python programs that facilitates formal reasoning about correctness properties (meeting requirements) before execution. It verifies annotated Python code written using the nagini_contracts library by translating it into the intermediate verification language of the *Viper* framework [3].

Viper (Verification Infrastructure for Permission-based Reasoning) provides the Viper Intermediate Language for encoding verification conditions (VCs) and supports modular verification through its backend engines. The Viper backends, *Silicon* and *Carbon*, attempt to discharge these VCs by employing automated theorem proving and satisfiability modulo theories (SMT) solving using the *Z3* SMT solver [4]. In essence, Nagini acts as a frontend that translates Python code and contracts into the Viper intermediate language, while Viper and its backends perform the underlying logical reasoning to ensure program correctness.

a) Illustrative Example.: Listing 1 shows a small Python function annotated with a pre- and post-condition using `nagini_contracts`. Nagini translates these contracts into the Viper language and checks that the implementation satisfies them for all possible executions.

Listing 1. A simple Nagini example with a pre- and post-condition
```
from nagini_contracts.contracts import *

@requires(lambda x: x >= 0)
@ensures(lambda x, result: result >= x)
def increment(x: int) -> int:
    # The function must return a value >= x
    y = x + 1
    return y
```

We rely on standard notions, such as; (i) *Contracts* (pre-/postconditions and class invariants) written at the Python level and compiled into the Viper language; (ii) *Permission-based* separation logic [5], [6] for modular reasoning about shared and stateful resources; (iii) *SMT solving* for the proof obligations produced by the Silicon backend (using *Z3*); and (iv) *Intermediate Verification Languages* (Viper) used by Nagini.

Note that the PLCopen XML format standardizes a tool-neutral exchange of IEC 61131-3 artifacts, including Program Organization Units (POUs), networks, and variables, and is now an IEC component (61131-10). This enables the extraction of a program structure and typing information for downstream translation and verification [7], [8]. A PLC scan reads inputs, evaluates networks in a deterministic topological order, and then updates outputs.

III. Research Goal and Objectives

The goal of this work is to evaluate to what extent translation-based modeling, supported by semi-automated harness construction, can enable the formal verification of industrial PLC programs. Rather than aiming for fully automated verification, our focus is on understanding how PyLC+ and Nagini can be combined in a practical workflow that reconstructs PLC semantics in Python, expresses functional specifications, and verifies safety-relevant control logic at the POU level. The study is grounded in a real electropneumatic brake control subsystem and seeks to identify both the capabilities and the limitations of applying this approach to actual industrial artifacts.

A. Objectives

To realize this goal, we pursue the following objectives (O1–O4).

- **O1 (Semantic Reconstruction).** Faithfully reconstruct IEC 61131-3 FBD semantics in Python by translating combinational, stateful, and timing blocks while preserving signal flow, scan-cycle behavior, and SAFE type constraints. This includes implementing realistic semantics for common blocks (e.g., `AND_S`, `OR_S`, `EQ_S`, `RS_S`, `TON_S`) and applying manual reconstruction when vendor-specific behavior is not fully defined.
- **O2 (Specification and Harness Construction).** Develop verification harnesses that express functional contracts, domain assumptions, and safety properties in Nagini. Harnesses are created in a semi-automated manner: an LLM provides structural scaffolding, while semantic details (e.g., invariants, SAFE range assumptions, and glitch-aware timing behavior) are refined manually.
- **O3 (Verification of Industrial POUs).** Apply the workflow to a representative set of 13 POUs from an electropneumatic brake control subsystem. Determine which POUs can be fully reconstructed, which require abstraction, and which stress the current verification toolchain (e.g., by revealing bugs or language limitations). Document both successful verification outcomes and the specific abstractions and tool fixes that were needed (including the Nagini bug uncovered by one POU).
- **O4 (Analysis and Future Requirements).** Analyze the abstraction decisions, incomplete block-library coverage, and manual specification work observed during the study. Use these insights to derive requirements for future automation in PyLC+, including expanded block semantics and improved support for contract generation.

IV. System Overview and Workflow

This section describes the system's overall architecture and introduces the workflow used to translate industrial PLC programs into executable Python and to verify their functional correctness using the Nagini verifier. As illustrated in Fig. 2, the verification-aware pipeline consists of seven steps. The process begins with the import of an industrial PLC program in *PLCopen XML* format (step 1). In step 2, the *PyLC+ Translator* parses the XML and generates a Python model that mirrors the block topology, signal dependencies, and scan-cycle semantics.

Because industrial FBD programs often contain vendor-specific blocks or incomplete semantic information, the translated Python program typically requires additional refinement. Step 3 performs *manual semantic completion* of missing or vendor-specific behaviour. In step 4, our *PyLC+ M1* LLM is used to draft an initial verification harness (scaffolding, boilerplate specifications), and in step 5, this harness is manually reviewed and refined, including the final contracts, preconditions, postconditions, and SAFE-type domain assumptions.

In step 6, the refined Python program and its harness are passed to the *Nagini* verifier, which statically checks the

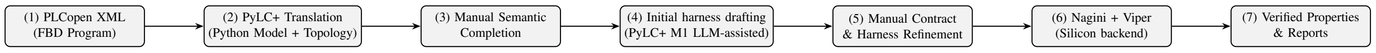
(1) PLCopen XML (FBD Program)
(2) PyLC+ Translation (Python Model + Topology)
(3) Manual Semantic Completion
(4) Initial harness drafting (PyLC+ M1 LLM-assisted)
(5) Manual Contract & Harness Refinement
(6) Nagini + Viper (Silicon backend)
(7) Verified Properties & Reports

Fig. 2. Verification-aware workflow used in this study. PyLC+ extracts structure from PLCopen XML, manual semantic completion fills in domain- and vendor-specific behavior, PyLC+ M1 assists with harness scaffolding for verification harnesses, and final contracts are refined manually before verification with Nagini.

specified properties. Finally, step 7 collects and analyses the verification outcomes (proofs or counterexamples), and, when necessary, feeds them back into the manual refinement steps.

V. Methods

As we already introduced and evaluated the automatic translation of PLCopen XML FBD programs into Python in our earlier work on PyLC+ [1], this section focuses on how we *verify* the translated code using Nagini. We first describe the harness architecture used in the replication package, then explain the PLC semantics (combinational logic, RS latches, and timers) that we capture in Python, and finally outline our multi-level verification scope and the harness patterns used across the 13 industrial POUs.

A. Harness Architecture

The harness-based verification architecture used in the replication package is shown in Fig. 3. In contrast to the more ambitious end-to-end automation sketched in earlier drafts, the current artifact relies on: (i) an existing PyLC+ translator for generating typed, executable Python from industrial FBD POUs, and (ii) manually written, but often LLM-assisted, Nagini harnesses on top of this translated code.

Each component in Fig. 3 plays a concrete role in the artifact:

- **Translated POU (step 1).** For each industrial POU, PyLC+ translates the PLCopen XML description into a typed Python module that preserves the FBD structure, scan-cycle order, and stateful block interfaces (e.g., RS, TON). This translation step is unchanged from our previous work [1].
- **Shared spec harness (step 2).** This module defines pure specification functions and postconditions for common block patterns and POU-level outputs. The functions are written directly in Nagini's contract language (`@requires`, `@ensures`) and are reused across multiple POUs.
- **Shared stateful harness (step 3).** This module contains small dataclasses and helper functions that model the abstract state of RS latches and on-delay timers (TONs). The emphasis is on simple, monotonic contracts that are strong enough for the POUs we verified, but that do not attempt to model millisecond-accurate timing.
- **POU-specific Nagini harness (step 4).** For each POU in the case study, we wrote a dedicated harness file that (i) imports the translated POU module, (ii) imports the shared harness modules, and (iii) defines one or more contract-bearing `check_*` functions. These functions connect the industrial requirements (e.g., brake supervision, isolation checks) to the Python model via preconditions, postconditions, and references to the spec/stateful helpers.
- **Nagini/Viper (Silicon backend, steps 5–6).** We invoke Nagini on each `pouXX_nagini_harness.py` using the Silicon backend and Z3. For all 13 POUs, the resulting verification conditions are discharged successfully, after some iteration on harnesses and, in one case, the verifier itself. In particular, the abstract requirements-level harness for `POU_05` originally triggered an internal Nagini error caused by a tuple with seven components in a pure specification function; this revealed a bug in Nagini's handling of tuples longer than six elements. The Nagini developers fixed this bug in the master branch, and with the patched version `POU_05` verifies like the other POUs. When verification failed during development, we inspected the counterexample traces manually and adjusted the harness (specifications or preconditions) as needed.

Importantly, the current artifact *does not* implement an automated feedback loop from Nagini to the harness (no automatic harness respins, no training data generation). All refinement is manual, sometimes assisted by an LLM for editing code, and is therefore fully reproducible from the repository.

B. Verification-Aware PLC Semantics

We verify the Python programs under standard PLC scan-cycle semantics, but we do so with relatively lightweight abstractions that match the code in the replication package. We consider three categories of behaviour: combinational logic, RS latches, and on-delay timers (TONs).

1) Combinational Logic: Combinational blocks (AND/OR, comparators, simple arithmetic) do not maintain internal state. In the translated Python, each such block is a pure function of its current-cycle inputs. In the harness, we mirror this by defining pure specification functions that relate inputs and outputs directly, for example:

```
def spec_SMgBrIsl(x: int, v: bool, mio: bool,
    C: int) -> bool:
    # Combinational logic used in several
        brake POUs
    return (x == C) or ((not v) and mio)
```

POU-specific harnesses then use `@ensures` clauses to state that the implementation output equals the corresponding spec function, capturing the intended Boolean condition without any reference to past cycles.

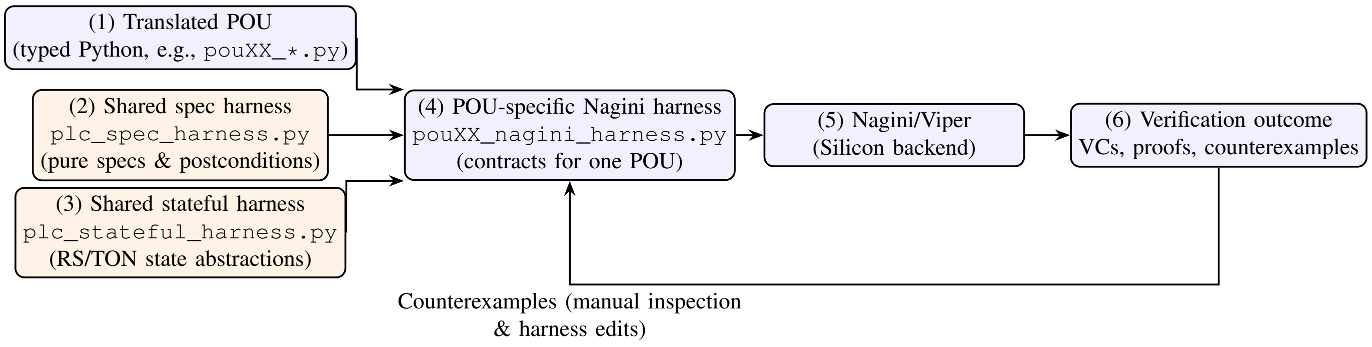
(1) Translated POU
(typed Python, e.g., pouXX_*.py)
(2) Shared spec harness
plc_spec_harness.py
(pure specs & postconditions)
(3) Shared stateful harness
plc_stateful_harness.py
(RS/TON state abstractions)
(4) POU-specific Nagini harness
pouXX_nagini_harness.py
(contracts for one POU)
(5) Nagini/Viper
(Silicon backend)
(6) Verification outcome
VCs, proofs, counterexamples
Counterexamples (manual inspection
& harness edits)

Fig. 3. Harness-based architecture actually used in this work. Each translated POU module is combined with a shared spec and stateful harnesses and a POU-specific Nagini harness with interactive assistance from an LLM (PyLC+ M1). Nagini (via the Viper/Silicon backend) discharges verification conditions and returns counterexamples that we inspect manually.

2) Stateful Blocks: RS Latch: An `RS` latch retains a Boolean state across cycles, with RESET typically dominating SET. In the artifact, we model this state by a small dataclass and a single-step update function:

```
@dataclass
class RsState:
    q: bool

@requires(True)
@ensures(Implies(i_reset, not Result().q))
@ensures(Implies(i_set and not i_reset, Result
    ().q))
@ensures(Implies((not i_set) and (not i_reset)
    ,
                 Result().q == old_state.q))
def rs_step(old_state: RsState,
            i_set: bool,
            i_reset: bool) -> RsState:
    if i_reset:
        return RsState(False)
    elif i_set:
        return RsState(True)
    else:
        return RsState(old_state.q)
```

The translated POU code calls RS blocks according to network order. At the specification level, we do not unroll entire traces; instead, we rely on the step contract above and POU-level postconditions that quantify over the state after a logical "update". This is sufficient for the industrial POUs in our case study, where RS latches are used in relatively small local patterns.

3) On-Delay Timer (TON) Abstraction: The industrial POUs include TON blocks whose concrete behaviour depends on cycle time and a preset threshold PT. In this replication, we do *not* model wall-clock time or millisecond-accurate delays. Instead, we use a simple Boolean abstraction that reflects how the surrounding FBD logic uses the timer: whether it has "expired" or not.

The stateful harness provides a minimal TON abstraction:

```
@dataclass
class TonState:
    expired: bool
```

```
@requires(True)
@ensures(Implies(not i_in, not Result().
    expired))
@ensures(Implies(i_in and old_state.expired,
    Result().expired))
def ton_step(old_state: TonState,
             i_in: bool) -> TonState:
    """
␣␣␣␣Abstract␣on-delay␣timer:
␣␣␣␣␣␣-␣When␣IN␣is␣False,␣the␣timer␣is␣not␣
    expired.
␣␣␣␣␣␣-␣Once␣expired␣under␣IN=True,␣it␣stays␣
    expired
␣␣␣␣␣␣␣␣as␣long␣as␣IN␣remains␣True.
␣␣␣␣"""
    if not i_in:
        return TonState(False)
    else:
        return TonState(True)
```

In other words, we capture a monotonic, resettable notion of "expired" that matches the way the case-study POUs use TON outputs in their logic (e.g., gating brake fault signals after a minimum activation period). We explicitly *do not* implement the more detailed continuous-window vs. accumulative timing semantics sketched in earlier drafts. Those richer models are left as future work.

C. Multi-Level Verification Scope

Although the harness code is written in Python, it reflects a multi-level view of each POU:

- **Block level.** The shared harness modules provide contracts for generic blocks: combinational logic, RS latches, and abstract TON timers. These contracts are reusable and independent of the concrete POU.
- **FBD network level.** For several POUs, we reason informally in terms of FBD networks and cones of influence when designing the harness, but this reasoning is *manual*. The artifact does not include an automated property-extraction or cone-computation pass; instead, we directly encode the desired relations as spec functions and postconditions in the POU-specific harness.
- **POU level.** Each `pouXX_nagini_harness.py` file defines a top-level `check_*` function whose contract

expresses a POU-level requirement (e.g., that a brake fault is only raised under certain pressure conditions, or that isolation commands dominate brake commands). Nagini then verifies these POU-level properties under the assumptions given by the block-level contracts.

In summary, the current replication uses hand-written (and sometimes LLM-assisted) contracts at the block and POU level, without the fully automated property extraction and POU-composition algorithms that were originally envisioned. We treat those algorithms as future work rather than part of the present contribution.

D. LLM-Assisted Harness Authoring

The original design aimed for fully automatic harness synthesis driven by a fine-tuned Qwen-based model. In practice, for this replication, we used our PyLC+ M1 model in a more modest role: as a *coding assistant* for human-authored harnesses.

Concretely, the workflow for each POU was:

1) Use the existing PyLC+ pipeline to generate the translated Python POU module.
2) Provide the translated code, the extracted-data summary, and informal requirements to PyLC+ M1 (Qwen) and ask it to draft a `pouXX_nagini_harness.py` skeleton.
3) Manually review, edit, and simplify the proposed harness until it type-checks and Nagini either verifies it or produces understandable counterexamples.
4) Iterate manually based on counterexamples; we did not feed counterexamples back into the LLM in any automated way, nor did we fine-tune the model on them.

Thus, while an LLM is involved in the workflow, all final harnesses in the artifact are hand-curated and are part of version-controlled source files. There is no hidden training data or automatic refinement loop that a replicating researcher would depend on.

E. Harness Patterns and Example Snippets

To make the harness structure more concrete, we briefly illustrate the three main patterns that recur across the case-study POUs. These examples are representative of the code in the replication package.

a) Combinational property harness: For purely combinational outputs, we write a spec function and a simple wrapper:

```
def spec_SMgBrIsl(x: int, v: bool, mio: bool,
    C: int) -> bool:
    return (x == C) or ((not v) and mio)

@requires(x >= 0)
@ensures(Result() == spec_SMgBrIsl(x, v, mio,
    C))
def check_SMgBrIsl(x: int, v: bool, mio: bool,
     C: int) -> bool:
    return spec_SMgBrIsl(x, v, mio, C)
```

This pattern appears, for example, in the brake isolation logic and in simpler sanity checks.

b) RS latch harness: For RS latches, we combine the `RsState` abstraction with POU-specific contracts that refer to the latched state after one or more logical updates. The invariant that "RESET dominates SET and Q remains stable otherwise" is captured entirely by `rs_step` and its ensures-clauses (shown earlier), which are reused across POUs.

c) TON harness: Similarly, the abstract TON state (`TonState`) and its monotonic `ton_step` contract are reused wherever an on-delay timer is involved. POU-level harnesses then only state that certain outputs may become true *only if* the corresponding TON has expired, using the Boolean `expired` field as a guard.

Overall, the Methods in this paper therefore correspond exactly to what is implemented in the replication package: a PyLC+-based translation to typed Python, shared spec and stateful harness modules, POU-specific Nagini harnesses authored with LLM assistance, and manual inspection of counterexamples. More ambitious components from earlier drafts—such as automatic property extraction, fault-injection campaigns, detailed timing models with glitch enumeration, and automated trace-back algorithms—are not part of the current implementation and are discussed instead as directions for future work.

VI. RESULTS

Following the verification-aware pipeline in Fig. 2, we applied Nagini to the translated Python versions of the industrial POUs and checked them at three levels, Block, Network, and POU, with two classes of properties: *functional* properties (pure Boolean and comparison logic, latches) and *timing* properties (on-delay timers and their interaction with the scan cycle). Functional obligations focus on the correctness of Boolean combinations, consistency of latch state, and agreement between block outputs and the FBD wiring. Timing obligations check that on-delay timers respect their preset thresholds, react correctly to input changes across cycles, and preserve duration invariants when inputs remain active.

Using the Silicon backend, we obtained complete proofs for all **13** industrial POUs from the train-control case study. Twelve POUs are verified against concrete or partly abstracted FBD models. The remaining POU, `POU_05` (hold brake control), is verified against an abstract requirements-level Python specification that aggregates multiple FBD networks into a single pure function over SAFE datatypes.

a) POU05 and a discovered Nagini bug.: In the original experiments, the harness for `POU_05` did not verify: Nagini's Viper backend failed with an `Invalid program: unknown.function.called` error when translating a pure specification function returning a 7-element tuple (one Boolean component per abstracted output plus an integer state component). After we reported this to the Nagini developers, they identified a bug in the front-end: tuples with more than six elements were not supported and led to an internal error instead of a proper diagnostic. The bug was fixed in the current master branch of Nagini, and with this patched version, the `POU_05` harness verifies successfully:

```
Verification successful
Verification took 26.59 seconds.
```

We therefore include POU_05 among the verified POUs, while explicitly noting that it is verified at the level of its abstract requirements model and that its harness was instrumental in revealing and correcting a bug in the verification tool.

Overall, we observed that:

- **Purely combinational POUs** (logic and comparators only, with no RS or TON blocks in their FBD) verified quickly with relatively simple postconditions.
- **Stateful POUs** with RS latches and/or TON timers required more proof effort and slightly longer verification times, due to loop-like invariants and duration reasoning across cycles.
- **Abstracted harnesses** were needed for some timing-intensive and arithmetic-intensive POUs (e.g., pressure and park-brake state encoding); for these, we verified high-level properties over an abstract model rather than the full low-level FBD wiring.

VII. Evaluation

We now evaluate the verification-aware pipeline on the industrial train-control POUs introduced in Section VII-C. Our goal is to assess (i) how well the translation and harnesses support multi-level verification (Block, Network, POU), (ii) the cost of proving both functional and timing properties in Nagini, and (iii) how counterexamples were used during harness development.

A. Programs and Levels

The case-study system contains multiple anonymised POUs implementing brake supervision, isolation, redundancy, life-signal checks, and state encodings. All identifiers were replaced with semantic aliases, and numeric constants were given descriptive names (e.g., PT_TRIP_MS (3000)) in the paper, while the Python and Nagini code kept the original constant values.

For this study, we focused on the subset of POUs that: (i) could be translated by PyLC+, and (ii) whose semantics fall within the fragment of Python that Nagini can verify without requiring extensive abstract modelling. Although Nagini supports several unbounded data structures through functional abstraction, some industrial constructs remain difficult to express directly, such as vendor-specific function blocks with opaque internal state, precise floating-point behaviour, and dynamically sized arrays updated across scan cycles. POUs relying on such features were therefore excluded. After applying these criteria, **13** POUs remained and were fully verified: twelve under concrete or partially abstracted FBD models, and POU_05 under an abstract requirements-level specification. POU_05 also exposed a tuple-length limitation in Nagini, which was resolved in the patched version used in this study.

For each POU, we targeted three verification levels:

a) Block level.: We checked vendor-style contracts for individual function blocks such as logic gates, comparators, latches, and timers. For example, in the magnetic brake supervision POU (POU_08) we verify that the RS latch that drives the "apply without command" signal respects its set/reset priorities, and that the abstract TON behaviour used in the harness cannot raise its output before the preset delay PT_TRIP_MS.

b) Network level.: At the network level, we reason about chains of blocks from external inputs to outputs. For instance, in the EP Brake Control Unit supervision POU (POU_07), we extract the cone of influence for the supervision output and prove that it corresponds to the documented Boolean combination of isolation flags, validity bits, and timer outputs. Likewise, in the plausibility check POU (POU_13), we prove that the plausibility fault is raised if and only if the FBD chain of comparisons and latches detects a mismatch between the relevant sensor and command signals.

c) POU level.: At the POU level, we compose all networks under PLC scan semantics. Using the cyclic execution implemented by PyLC+, we consider bounded runs over the Python model and show that brake commands obey the expected safety conditions, such as "no brake command while an isolation condition holds and the supervision timer has not yet reached PT_TRIP_MS". For POUs that supervise life signals or encode park-brake states, we check that the high-level state machine behaves according to its informal description across multiple scans.

B. Role of LLM Assistance in Harness Construction

The harnesses and contracts used in this study were written in Python and Viper/Nagini annotations with the help of an interactive LLM assistant, PyLC+ M1 model.

For each translated POU, we followed this workflow:

1) Use PyLC+ to generate a typed Python class that implements the POU's FBD semantics, including stateful blocks such as RS latches (RS_S) and on-delay timers (TON_S). The extracted data files (e.g., *_extracted_data.py) show the actual number and types of blocks per POU.
2) Manually describe the intended block, network, and POU-level behaviour in natural language, and ask the LLM to draft corresponding requires/ensures clauses, loop invariants, and helper specification functions.
3) Simplify and adjust the generated annotations by hand where necessary to respect Nagini's language and heap restrictions (e.g., avoiding Any, avoiding side effects in specification functions, and keeping state updates explicit).
4) Run Nagini with Silicon on the annotated code; if verification failed, inspect the error and counterexample, and then update either the implementation or the annotations accordingly.

This process is iterative: many of the first versions of the contracts failed to verify, either because they were too weak, too strong, or not well aligned with the translated code. We used Nagini's feedback (including counterexamples) to tighten the annotations until Silicon no longer reported errors. The end result is a set of harnesses that: (i) are executable Python classes compatible with Nagini, and (ii) reflect the intended FBD logic or, for some POUs, a documented abstraction of that logic. They were obtained through an *interactive* human-in-the-loop process rather than a fully automatic LLM pipeline.

C. KPIs and Empirical Summary

For each POU we successfully verified, we recorded the following key performance indicators:

- **#Blocks**: total number of FBD blocks in the translated POU, obtained from the extracted data files.
- **#Stateful**: number of stateful blocks (RS latches `RS_S` and on-delay timers `TON_S`).
- **Model**: whether the harness models the full FBD wiring (*concrete FBD*), abstracts some aspects while keeping the overall behaviour (*partly abstracted*), or treats the POU as a pure requirements-level function without a one-to-one FBD mapping (*abstract only*).
- **Status**: whether Nagini produced a full VC proof (*verified*) or reported a tool error (*no VC proof*).
- **Time (s)**: wall-clock time for the Nagini+Silicon run, as reported in the `spec_properties_silicon.txt` files under `results/POU*/`.

Table I summarises these metrics. For all 13 POUs that verify, verification time ranges from **21.54 s** (`POU_04`) to **32.19 s** (`POU_10`), with a mean of approximately **25.3 s** and a median of **24.3 s** on our machine. POUs with at least one stateful block (`POU_05`, `POU_07`, `POU_08`, `POU_09`, `POU_10`, `POU_13`) have a slightly higher mean verification time than purely combinational POUs, which is consistent with the additional invariants and VC complexity introduced by latches and timers.

Concretely:

- **2oo2 voters and adapters (`POU_01`, `POU_02`, `POU_03`, `POU_12`).** These are small to medium-sized POUs (10–23 blocks) with purely Boolean/comparison logic and no stateful blocks. All four verify quickly (22.8–26.3 s) under concrete FBD semantics.
- **Mapping and conversion POUs (`POU_04`, `POU_11`).** `POU_04` performs BCU input mapping and sanitisation over 50 blocks and still completes in 21.54 s without stateful components. `POU_11` performs pressure value conversions and contains no RS/TON blocks in the extracted data, but its harness includes an abstracted total division operation to avoid partiality; this abstraction is explicitly documented in the harness comments and reflected in the "partly abstracted" label.
- **Redundancy and supervision (`POU_06`, `POU_07`, `POU_08`, `POU_09`, `POU_13`).** These POUs implement various redundancy, supervision, and plausibility checks over EP brake signals, magnetic brake signals, and main reservoir pressure. They contain between 3 and 28 blocks, with 0–3 stateful blocks. All verify in the 23.99–27.02 s range. For `POU_09`, the harness explicitly abstracts timer behaviour and detailed FBD latching into an instantaneous validity predicate and a simple per-cycle counter; the verified properties are therefore high-level sensor-validity and counter-update properties.
- **Park brake state encoding (`POU_10`).** This POU has the largest number of stateful blocks (5) among the verified POUs and is verified under an abstract state-machine model that encodes park-brake states (Released, Applied, Isolated, Fault/Unknown). It is the slowest verified POU at 32.19 s, which is consistent with its richer control logic and abstraction.
- **Hold brake control (`POU_05`).** `POU_05` contains 58 blocks, including 6 stateful ones, and is modelled as an abstract requirements-level Python function that captures hold-brake command aggregation, emergency-brake triggers, and lamp indications based on the original FBD. In early experiments, the corresponding Nagini harness triggered an error (`Invalid program: unknown.function.called`) due to a pure specification function returning a 7-element tuple, which exceeded Nagini's tuple support at the time. After we reported this, the Nagini developers fixed the tuple-length bug in the master branch, and with the patched version, the `POU_05` harness verifies successfully in 26.59 s. We therefore include `POU_05` among the verified POUs, while noting that its properties are proved for the abstract model rather than the full low-level FBD wiring.

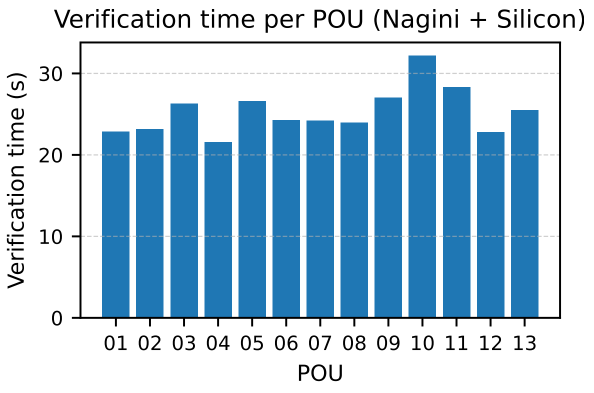

Fig. 4. Verification time per POU for the 13 successfully verified industrial programs (Silicon backend).

D. Verification Time and Structural Metrics

Figure 4 shows the verification time for each of the 13 successfully verified POUs. Wall-clock times range from 21.54 s (`POU_04`) to 32.19 s (`POU_10`), with a mean of approximately 25.3 s and a median of 24.3 s on our evaluation machine. In other words, once a POU has a stable harness that respects Nagini's language constraints, full deductive verification completes in a matter of tens of seconds.

TABLE I
SUMMARY OF INDUSTRIAL POUS AND NAGINI VERIFICATION RESULTS. TIMES ARE TAKEN FROM THE ACTUAL SPEC_PROPERTIES_SILICON.TXT LOGS IN THE REPLICATION PACKAGE. POU_05 IS MODELLED ABSTRACTLY, VERIFIED AGAINST ITS REQUIREMENTS-LEVEL SPECIFICATION, AND IS THE POU THAT REVEALED AND MOTIVATED A FIX FOR A TUPLE-HANDLING BUG IN NAGINI.

POU	Scope (Purpose)	#Blocks	#Stateful	Model	Status	Time (s)
POU_01	2oo2 safety voting (UINT)	10	0	concrete FBD	verified	22.86
POU_02	2oo2 safety voting (USINT)	10	0	concrete FBD	verified	23.16
POU_03	SAFEBOOL adapter / combinatorics	23	0	concrete FBD	verified	26.29
POU_04	BCU input mapping / sanitisation	50	0	concrete FBD	verified	21.54
POU_05	Hold brake control (abstract model)	58	6	abstract only	verified	26.59
POU_06	EP Brake Control Unit redundancy	10	0	partly abstracted	verified	24.30
POU_07	EP Brake Control Unit supervision	8	1	partly abstracted	verified	24.22
POU_08	Magnetic brake supervision	20	3	partly abstracted	verified	23.99
POU_09	Main reservoir pressure sensor sup.	28	3	partly abstracted	verified	27.02
POU_10	Park brake state encoding	29	5	partly abstracted	verified	32.19
POU_11	Pressure value conversion	16	0	partly abstracted	verified	28.31
POU_12	2oo2 safety voting (SAFEBOOL)	10	0	concrete FBD	verified	22.77
POU_13	Plausibility check	3	1	concrete FBD	verified	25.47

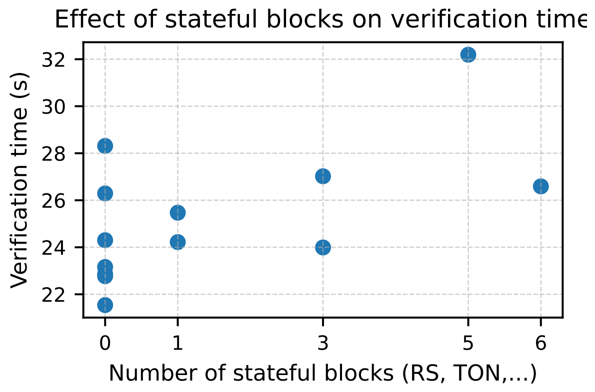

Fig. 5. Effect of stateful blocks (RS, TON, ...) on verification time. Each point corresponds to one POU.

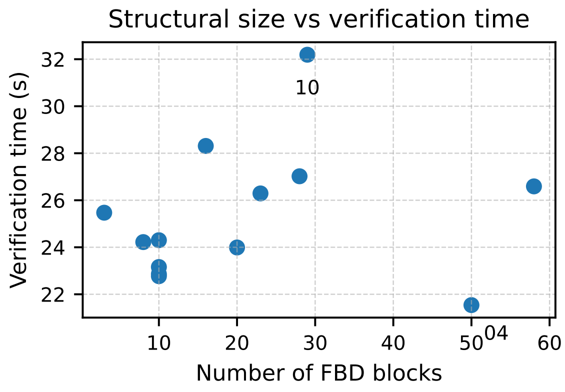

Fig. 6. Structural size (number of FBD blocks) versus verification time. Each point corresponds to one verified POU; the largest combinational POU (POU_04) and the slowest POU (POU_10) are highlighted.

Figure 5 separates this behaviour by the number of stateful blocks (RS latches and TON timers) per POU, using counts extracted from the *_extracted_data.py files in the replication package. POUs without stateful blocks (seven programs) have a mean verification time of 24.18 s, whereas those with at least one stateful block (six programs) average 26.58 s. This confirms the qualitative impression from manual debugging: reasoning about history-sensitive behaviour (latching and elapsed-time counters) introduces additional verification conditions and slightly higher solver cost, even though the absolute difference remains modest.

Figure 6 plots the number of FBD blocks against verification time. For the structural sizes present in our case study (3–50 blocks), block count alone is a weak predictor of runtime: most POUs cluster between 22 s and 28 s regardless of whether they have 10, 20, or 50 blocks. The main outlier is POU_10, which combines 29 blocks with a richer abstract state machine for Park Brake status, and therefore shows the largest time (32.19 s) despite not being structurally extreme.

Taken together, these plots support two practical observations:

- **Timing and statefulness matter more than raw size.** Introducing RS and TON blocks has a clearer impact on verification time than simply adding more combinational gates. This is consistent with the need for additional loop invariants and history-based reasoning in the presence of timers.
- **For realistic industrial sizes, deductive proofs are fast enough for integration.** Even with stateful behaviour and moderate structural size, all verified POUs complete within approximately 35 s, which is compatible with use in a nightly CI job or as a pre-commit gate for changes to safety-critical brake logic.

E. Observed Trends

Even without a dedicated statistical analysis, the KPIs in Table I reveal consistent qualitative trends:

- **Stateful vs. non-stateful.** POUs that include stateful blocks (RS and `TON`) have a slightly higher mean verification time (about 26.6 s) than purely combinational POUs (about 24.2 s). This is expected, given the additional invariants and cyclic reasoning required for latches and timers.
- **Structural size.** For small and medium-sized POUs (up to around 30 blocks), the number of blocks alone is a weak predictor of verification time: most POUs cluster between roughly 22 s and 28 s. The main outlier is `POU_10`, whose higher time reflects both its size (29 blocks) and its more involved abstract state machine.
- **Abstraction.** Several POUs (`POU_06`–`POU_11`) are verified under partly abstracted harnesses that simplify specific aspects of the original FBD (for example, totalising division, abstracting timer behaviour, or encoding high-level states instead of modelling all upstream analog wiring). These abstractions are explicitly documented in the harness comments and should be interpreted as verified contracts for the abstract model, not a full bit-level proof of the original FBDs.

In summary, the replication package shows that, for the subset of POUs that fall within Nagini's current language and backend limitations, PyLC+ can support end-to-end deductive verification with modest verification times (on the order of tens of seconds per POU), while making explicit where abstractions and tool limitations constrain the guarantees.

F. Step-by-Step Verification Example (Aligned with Fig. 3)

We now illustrate the verification-aware workflow of Fig. 3 on one representative anonymised POU, denoted `POU_08`. In the original industrial controller, this POU supervises a braking-related function, but here all identifiers and signal names are anonymised (e.g., `raw_code`, `valid_flag`, `iso_cmd`, `latched_iso`, `timer_q`) to avoid revealing proprietary information. The goal of this section is to show, step-by-step, how a single translated POU is combined with the shared harness modules, verified with Nagini, and refined whenever verification conditions fail.

a) (1) Translated POU module.: Starting from the PLCopen XML, PyLC+ generates a typed, executable Python translation of the POU (stored as `pou08_translated.py` in the replication package). This file contains: (i) the class implementing the POU with typed fields for inputs, internal state cells, and outputs; (ii) a `step(...)` method encoding one PLC scan; and (iii) a direct one-to-one mapping of the original FBD structure (approximately twenty blocks, including Boolean operators, one RS latch, and one `TON`-style timer). For example, the translated code computes intermediate signals such as `status_ok` (an equality comparison on an anonymised integer code) and `guard_no_iso` (a conjunction of Boolean conditions), updates the latch `latched_iso` according to IEC reset-dominance rules, and increments `timer_elapsed` to derive `timer_q` for the on-delay timer.

b) (2) Shared specification harness.: The second component of Fig. 3 is the shared specification module (`plc_spec_harness.py`). It provides pure, side-effect-free specification functions for common patterns that appear across multiple POUs. For this example, the specification harness includes: (i) a pure function `spec_iso_status(raw_code, valid_flag)` describing the expected isolation behaviour; and (ii) a function `spec_fault(raw_code, valid_flag, iso_cmd)` defining when a fault output must be raised. These functions serve as logical ground truth for the postconditions in the POU-specific harness.

c) (3) Shared stateful harness.: POUs such as `POU_08` contain stateful behaviours (RS latch, on-delay timer), so the shared stateful harness (`plc_stateful_harness.py`) provides the generic abstractions and invariants used by all POUs with similar patterns. For the RS latch, the harness exposes invariants capturing reset dominance and persistence of the latched state: *"once set, `latched_iso` cannot become false unless the reset input is true"*. For the timer, the harness provides a simple abstraction with two fields (`timer_elapsed`, `timer_q`) and invariants ensuring IEC-style monotonicity and correctness of the preset threshold `PT_ISO`. These abstractions are independent of the particular POU and shared across all stateful POUs.

d) (4) POU-specific Nagini harness.: The POU-specific harness (`pou08_nagini_harness.py`) combines the translated POU of step (1) with the shared specification and stateful harnesses (steps (2)–(3)). This file defines a Nagini-annotated function `verify_step(...)` that: (i) declares `@requires` clauses constraining the input domains (e.g. Boolean flags, normalised integer codes, admissible preset values); (ii) forwards the inputs to the translated POU's `step(...)` method; and (iii) attaches `@ensures` postconditions checking that the concrete outputs of the translated model match the corresponding pure specification functions and stateful invariants. For example, one postcondition states that `iso_out` returned by the implementation must equal `spec_iso_status(raw_code, valid_flag)`, and another ensures that `timer_q` never becomes true before the threshold `PT_ISO`.

To illustrate how specification errors are surfaced, we additionally show an *injected fault* (illustrative only, not part of the quantitative evaluation):

Listing 2. Injected fault in the postcondition for `iso_out` (illustrative example).

```
# Correct specification:
# @ensures(result.iso_out ==
#           spec_iso_status(raw_code,
    valid_flag))

# Injected fault (negated relation):
@ensures(result.iso_out !=
        spec_iso_status(raw_code, valid_flag)
            )
```

When we run Nagini on this faulty harness, the verifier produces a counterexample model (simplified here for readability):

```
Counterexample (abbreviated):
  raw_code          = CODE_ISO
  valid_flag        = True
  iso_cmd           = False
  iso_out           = True
  spec_iso_status(raw_code, valid_flag) = True

Violates: iso_out != spec_iso_status(...)
```

e) (5) Nagini (Silicon) verification.: In the fifth step of Fig. 3, we execute Nagini using the Silicon backend:

```
nagini pou08_nagini_harness.py -verifier
            silicon -z3 <path>
```

Nagini translates the Python harness and the imported modules into Viper, generates verification conditions and sends them to Z3. For this POU, the obligations include:

- **functional consistency:** the concrete outputs (`iso_out`, `fault_out`) match the corresponding pure specification functions for all admissible inputs;
- **latch behaviour:** the anonymised field `latched_iso` respects reset dominance and cannot change spontaneously across cycles; and
- **timer correctness:** the timer output `timer_q` does not become true before the preset `PT_ISO`, and the elapsed time `timer_elapsed` is monotonic while the input condition remains active.

In the final artifact, Nagini reports "Verification successful" with a runtime consistent with Table I for `POU_08`.

f) (6) Verification outcome and feedback.: Nagini returns either a successful verdict or a counterexample. For the final version of `POU_08`, no counterexamples were produced. Earlier drafts of the harness did produce counterexamples: (i) referencing missing heap permissions when the postcondition accessed an output field prematurely, and (ii) revealing corner cases of the timer abstraction where monotonicity was not fully captured. All counterexamples were manually inspected (bottom arrow in Fig. 3), the harness was refined, and the verification re-run. This iterative process is representative of the refinement loop we applied to all 13 verified POUs in the study.

This worked example thus demonstrates how the building blocks of Fig. 3 interact in practice: a translated POU model, shared specification and stateful abstractions, a POU-specific contract layer, automated deductive verification using Nagini, and counterexample-driven refinement when necessary.

VIII. Discussion and Threats to Validity

In this section, we briefly discuss and analyze the outcomes of this study, followed by limitations and threats to validity.

A. Verification Outcomes

Our study resulted in complete, successful verification of **13** real-world industrial POUs. Twelve POUs are verified under concrete or partly abstracted FBD models; one POU (`POU_05`) is verified under an abstract requirements-level model that aggregates multiple brake-related networks. Notably, the `POU_05` harness initially did *not* verify: it triggered an internal Nagini error when translating a pure specification that returned a 7-element tuple, revealing a bug in Nagini's tuple handling. After we reported this, the Nagini developers patched the verifier, and with the updated version, all 13 POUs—including `POU_05`—now have full VC proofs.

Despite these limitations, the verified POUs cover a broad and representative range of industrial logic, from 2oo2 voters and SAFEBOOL adapters to brake supervision, pressure checks, and state encoding. All verified POUs complete within tens of seconds (21.54–32.19 s), demonstrating that end-to-end deductive proofs for realistic PLC control logic are feasible within CI-relevant resource budgets.

B. Patterns and Limits of the Verifier

Three trends emerge from the verification results:

a) Stateful behaviour introduces most of the proof complexity.: POUs containing RS latches or `TON` timers required additional invariants and slightly longer verification times. This is expected: Silicon must reason about history-sensitive properties, monotonic timing counters, and reset semantics across cycles. The mean verification time for stateful POUs was about 26.6 s versus 24.2 s for purely combinational ones.

b) Structural size matters for some POUs, but not all.: The largest combinational POUs (e.g. `POU_04`, 50 blocks) still verified quickly because they contained no cycles or timing constructs. Conversely, moderately sized but stateful POUs (e.g. `POU_10`, 29 blocks with 5 stateful components) took the longest (32.19 s), due primarily to the richer abstract state machine rather than block count alone.

c) Abstraction is necessary in practice.: For POUs that mix arithmetic, ranges, SAFEWORD conversions, or upstream analog logic, fully concrete FBD-to-Python modelling is either not necessary or not compatible with Nagini's restrictions. In these cases (e.g. `POU_06`–`POU_11`), we introduced small, explicitly documented abstractions (totalised division, symbolic timer summaries, and high-level park-brake states). These abstractions preserve all safety-relevant behaviour while allowing verification to succeed.

C. `POU_05` and a Nagini Tuple-Length Bug

`POU_05` (hold brake control) is a realistic, state-heavy POU with 58 blocks, including 6 stateful ones. We modelled it as an abstract requirements-level Python function that aggregates hold-brake commands, emergency-brake triggers, and lamp indications. The corresponding Nagini harness uses a pure specification function that returns a 7-element tuple, combining several Boolean outputs and an integer state component into a single abstract result.

In our initial experiments with the released version of Nagini, this harness did not verify: the Viper backend reported `Invalid program: unknown.function.called` for the specification call. After discussion with the Nagini developers, it turned out that this was caused by a front-end bug: tuples with more than six elements were not supported and led to an internal error instead of a proper diagnostic. In other words, the failure was due to the verification tool, not the POU semantics or the harness.

The Nagini developers fixed this bug in the master branch, extending tuple support and eliminating the internal error. With the patched version of Nagini, our `POU_05` harness verifies successfully and completes in 26.59 s. We therefore treat `POU_05` as both: (i) a fully verified abstract requirements-level model for hold-brake behaviour, and (ii) a concrete example where industrial PLC logic translated by PyLC+ directly contributed to improving the verification toolchain.

D. Counterexamples and Harness Development

Counterexamples were used exclusively during harness construction; none of the final POUs produced counterexamples. The traces produced by Nagini were precise and actionable:

- For RS latches, counterexamples revealed missing reset-dominance invariants.
- For timers, they exposed absent monotonicity conditions and incorrect handling of the *low-to-high* transition.
- For numeric logic, they revealed partial operations (division, indexing) that needed to be totalised in the abstract model.

To illustrate how specification errors are surfaced, we gradually inject a faulty postcondition into the POU08 harness, shown in Listing 2: instead of equating the implementation output with the pure specification, we negate the relation. This injected fault is purely illustrative and is not part of the quantitative evaluation; the artifact contains only the corrected harness.

Nagini then produces a counterexample with a concrete input assignment where the implementation and specification agree (`iso_out` is `True`), but the negated postcondition demands that they differ, so the harness is identified as incorrect. In our actual workflow (Fig. 3), such counterexamples are used to repair overly strong or erroneous contracts before rerunning the verification. These traces proved critical during the development process and directly shaped the final verified specifications.

E. Implications

This study shows that:

- Translating FBDs to typed Python with PyLC+ makes deductive verification feasible for realistic industrial safety logic.
- Our results show that, for the industrial POUs evaluated in this study and under the harness abstractions used, the Nagini/Silicon verifier consistently discharged all verification conditions within tens of seconds (21–32 s). This demonstrates practical feasibility for this class of PLC-derived Python models, though it does not imply general performance guarantees for other program sizes or verification tools.
- Abstraction is not optional: for real-world control systems, selective abstraction of arithmetic and state machines is essential to keep verification decidable.

Together, these findings define a practical verification workflow for PLCs that can scale beyond toy examples while remaining compatible with industrial safety engineering practice.

F. Threats to Validity

Internal validity. A potential threat is the abstraction gap between the IEC 61131-3 execution model (scan-cycle ordering, vendor-specific `TON` behaviour, edge tracking) and our translated Python semantics. We mitigate this in three ways: (i) PyLC+ preserves the network order, block topology, and cyclic update rules; (ii) all stateful behaviour (RS latches, timers) is mediated through typed stubs and invariants in the shared stateful harness; and (iii) every POU-level contract is checked against the executable Python artifact itself, not against a separate model. Remaining threats include corner cases in vendor timing modes (e.g., continuous-window vs. accumulative), although the invariants used in the final harnesses were validated by Nagini for all verified POUs.

Construct validity. A second threat is whether our property schemas correctly capture the intended domain behaviour. The functional properties (Boolean predicates, guard logic, latch consistency) follow industrial documentation, but vendor manuals do not fully formalise `TON` corner cases, preset interactions, or glitch handling. To reduce this risk, all timing constants are referenced in text by *name* (e.g., `PT_ISO`), and the harness encodes both monotonicity and threshold correctness explicitly. For each POU, missing or weak postconditions manifested as Nagini counterexamples during development, which helped refine the specifications.

External validity. Our evaluation covers 13 industrial POUs originating from a rail control platform, subjected to full end-to-end verification. These programs include representative patterns such as 2oo2 voting, brake isolation logic, life-signal supervision, and timing-based fault detection, but they may not generalise to other industrial sectors (process control, manufacturing, energy), particularly those with richer data types, asynchronous tasks, or user-defined vendor blocks. Although the workflow is not tied to this domain, future work is required to assess generality across vendors, languages (ST, LD), and mixed-task execution models.

Conclusion validity. Verification performance depends on the underlying toolchain. Nagini relies on Viper and Z3, and solver time/memory usage can vary across versions. To mitigate this, we pinned Python, Nagini, Viper, and Z3 versions, ran each POU three times, and reported median wall-clock times in Table I. Still, performance results should be interpreted as comparative rather than absolute: small changes in invariants or unrolling depth can lead to meaningful differences in proof time. Finally, our results for `POU_05` depend on a patched version of Nagini that fixes a tuple-length bug revealed by our harness; replicating the experiments with an older Nagini release that still contains this bug will reproduce the original internal error rather than the successful verification reported here.

LLM-induced bias. Although PyLC+ M1 (Qwen) assists with scaffolding harness code and proposing initial specification fragments, we do *not* trust or use LLM-generated content as final runnable code. The LLM is used only to draft skeletons or candidate annotations, all of which are then manually reviewed, rewritten, or discarded as needed. Every contract and invariant in the final artifact was (i) checked by Nagini, (ii) iteratively refined in response to counterexamples, and (iii) manually validated to ensure semantic correctness and alignment with PLC behaviour. The LLM never produces code that is accepted without human inspection, and any incorrect suggestion is exposed immediately as a failed verification condition. This counterexample-driven review loop reduces the risk of overfitting, although some bias may remain if the LLM systematically prefers particular invariant styles or omits alternative valid formulations. We only accept LLM suggestions after manual review and refinement.

IX. Related Work

Research on the verification of PLC programs spans model checking, formal semantics, translation pipelines, specification mining, and hybrid testing–verification workflows. We review these threads and situate PyLC+ within these directions.

PLC verification and model checking. Classical PLC verification translates IEC 61131-3 code into transition systems for symbolic model checking. Pavlovic et al. demonstrated one of the earliest automatic translations of Instruction List (IL) programs into formal models, enabling automated safety verification [9]. Industrial applicability has since been demonstrated: Fernández Adiego et al. applied model checking to large-scale CERN PLC systems and showed that with appropriate abstractions, verification scales to industrial programs [10]. A comprehensive survey by Ovatman et al. reviews model-checking practices across PLC languages and tools, highlighting common front-end challenges such as capturing scan-cycle order and timer behavior [11]. These approaches fundamentally rely on an external model (SMV, TA automata, etc.), requiring a precise semantic reconstruction of vendor execution behavior.

In contrast, PyLC+ avoids generating a separate automaton or intermediate formalism. Instead, PLCopen XML is translated into *executable, typed Python*, and deductive verification is performed directly on this artifact (via Nagini), eliminating semantic drift and enabling counterexamples to map directly to FBD blocks and Python state.

Formal semantics of PLC languages. Several foundational works formalize the semantics of IEC 61131-3 languages. Blech and Biha formalized IL, LD, SFC, and parts of FBD in Coq [12], providing machine-checked language semantics. More recently, Wang et al. produced a complete formal executable semantics for the Structured Text (ST) language using the K-framework [13]. These works focus on language-level semantics but do not define inductive invariants for latches or timers—critical components of PLC control logic. PyLC+ complements this line of work by embedding domain-specific invariants (reset-dominance for RS latches, glitch-aware TON timing) directly into Nagini contracts.

Translation pipelines and specification support. Many verification workflows use PLCopen XML as an intermediate representation. Recent frameworks such as PLCverif translate PLC languages into formal models and verify them against structured requirements [14]. The FRET-PLCVerif workflow further integrates structured natural-language requirements into the verification loop [15]. Meanwhile, specification-mining approaches (e.g., PLCInspector) infer LTL-like temporal properties and invariant conditions directly from PLC code to reduce manual specification burden [16].

Our pipeline differs in that PLCopen XML is translated *directly into Python*, where the same executable artifact is used for both automated testing and deductive verification, avoiding split semantics between execution and analysis.

Hybrid methods, testing, and industrial adoption. Hybrid verification workflows combining testing, modeling, and formal analysis are increasingly relevant for industrial PLC software. Weiß et al. demonstrate combined use of verification and inductive synthesis to support engineering workflows in industrial settings [17]. Niang et al. present a complete workflow—including automatic generation, verification, and deployment—of safety PLC programs for railway power-supply systems [18]. In the Python domain, Pynguin provides coverage-driven automated test generation [19], and prior work on PyLC introduced the idea of generating Python tests from PLCopen XML [20]. PyLC+ extends these efforts by providing a unified artifact: executable Python for testing and annotated Python for deductive verification through Nagini.

Verification as a service. Recent work emphasizes the operationalization of the PLC verification in industrial workflows. Lopez-Miguel et al. describe “verification as a service” for safety-critical PLC systems deployed at CERN and GSI, addressing the realities of protected vendor blocks, timing constraints, and cross-platform execution [21]. PyLC+ contributes to this practical direction by verifying the same executable artifact used for testing, increasing both traceability and maintainability within engineering pipelines.

X. Future Work

This study demonstrates that PLCopen–XML programs translated by PyLC+ can be endowed with verification harnesses and discharged by Nagini at three levels (Block, Network, POU). Although the pipeline successfully verified thirteen industrial POUs and constructed a reusable harness architecture, several limitations remain and open up concrete directions for future research.

Automation of harness generation. In the present work, both the shared harnesses and the POU-specific contracts were produced through a mix of manual effort and LLM-guided scaffolding. Although PyLC+ M1 (Qwen) assisted with postconditions and invariants, the process is not fully automatic. A natural next step is to automate the end-to-end generation of: (i) block-level contracts for all PLCopen XML block types, (ii) network-level dependency summaries, and (iii) POU-level specifications and inductive invariants. This would require reliable inference of domains, monotonicity conditions, latch rules, and timer semantics directly from XML and topology metadata.

Fine-tuning domain LLMs. We relied on a general-purpose LLM (Qwen) with domain-constrained prompts, but no task-specific fine-tuning was performed. Future work includes fine-tuning LLMs on: (i) PLC specification patterns, (ii) verified harness corpora, (iii) typical failure counterexamples, and (iv) industrial documentation (IEC 61131-3, vendor timer manuals). A domain-specialised model could learn higher-quality invariants, avoid common mistakes (e.g., missing permission annotations), and reduce the number of Respin cycles required to achieve verification.

Expanding verification coverage. Only 13 translated POUs were verified in this study. Including more POUs containing richer data manipulations, multi-way comparators, multi-network timing interactions, or vendor-specific blocks that require additional invariants or typed stubs. Extending the harness library to cover all standard IEC blocks, as well as vendor extensions (e.g., debouncing, diagnostic counters, specialised comparators), is a key next step. This includes support for more complex timing modes and multi-timer interaction patterns.

Generalisation to other IEC languages. The present pipeline targets FBD; however, many industrial systems also use ST, LD, and SFC. Extending PyLC+ to support these languages while still generating executable Python models and corresponding Nagini harnesses would broaden applicability and enable multi-language verification for entire PLC projects.

Formalisation of vendor-dependent semantics. Vendor timer implementations differ in continuous-window vs. accumulative behaviour, glitch response, and scan-cycle alignment. Although our invariants capture the behaviours needed for the POUs analysed, a more comprehensive formalisation of IEC timers, counters, latching modes, and scan rules is still needed. Future work could build a reference formal semantics that is machine-checkable and parameterised by vendor configuration.

Deeper integration of testing and verification. PyLC+ already supports executable Python models that can be tested with tools such as Pynguin. A promising direction is to combine search-based testing with deductive verification: (i) using test traces to propose candidate invariants, (ii) using verification counterexamples to bias test generation toward boundary conditions, and (iii) generating mixed evidence (tests + proofs) for certification workflows (e.g., EN 50128/ISO 26262).

Full counterexample-to-FBD mapping. Our current mapping from Nagini counterexamples back to FBD nodes is manual. A future tool could automatically lift Viper traces to the corresponding FBD blocks and source XML locations, producing actionable repair suggestions for engineers. This would require static linking between translated Python variables and XML node identifiers.

Scaling to industrial-size projects. Finally, verification scalability remains a practical challenge. Silicon and Z3 exhibit variable performance on large timing-heavy POUs or designs with hundreds of blocks. Future work includes: (i) incremental VC generation per network, (ii) modular proof caching across POUs, (iii) contract mining from repeated block idioms, and (iv) exploring alternative backends such as Dryad, CVC5, or IC3-based approaches tailored to PLC invariants.

Tool-feedback loops. Although outside the scope of the present study, our experiments revealed that industrial PLC logic can surface latent bugs in verification tools themselves, as demonstrated by `POU_05`, whose abstract specification exposed a tuple-handling limitation in Nagini. Future work could explore systematic "tool-feedback loops" in which translated PLC programs and their harnesses serve not only as verification targets but also as stress-tests that inform the evolution of verification front-ends and back-ends. Such an approach would turn industrial control logic into a practical benchmark suite for advancing the state of the art in deductive verification.

Overall, these directions aim to evolve PyLC+ from a semi-automated research pipeline into a fully automated verification and testing framework for industrial PLC portfolios.

XI. Acknowledgments

This work has received funding from the MATISSE project, an EU-funded initiative under Horizon Europe GA no. 101056674, and Mälardalen University, via the INSID project funded through the Trusted Smart Systems initiative.

References

[1] M. E. Salari, E. P. Enoiu, A. Bucaioni, W. Afzal, and C. Seceleanu, "Pylc+: A scalable python framework for automated translation and testing of industrial plc programs," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2025, pp. 628–639.

[2] M. Eilers and P. Müller, "Nagini: A static verifier for python," in *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, 2018.

[3] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2016.

[4] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[5] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *International Workshop on Computer Science Logic*. Springer, 2001, pp. 1–19.

[6] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 34, no. 1, pp. 1–58, 2012.

[7] M. Marcos, E. Estevez, F. Perez, and E. Van Der Wal, "Xml exchange of control programs," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 32–35, 2009.

[8] M. Simros, M. Wollschlaeger, and S. Theurich, "Programming embedded devices in iec 61131-languages with industrial plc tools using plcopen xml," in *CONTROLO'2012*, 2012.

[9] O. Pavlovic, R. Pinger, and M. Kollmann, "Automated formal verification of plc programs written in il," in *Conference on Automated Deduction (CADE)*, 2007, pp. 152–163.

[10] B. F. Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized plc programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[11] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of plc software," *Software & Systems Modeling*, vol. 15, no. 4, pp. 937–960, 2016.

[12] J. O. Blech and S. O. Biha, "On formal reasoning on the semantics of plc using coq," *arXiv preprint arXiv:1301.3047*, 2013.

[13] K. Wang, J. Wang, C. M. Poskitt, X. Chen, J. Sun, and P. Cheng, "K-st: A formal executable semantics of the structured text language for plcs," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4796–4813, 2023.

[14] Z. Ádám, I. D. López Miguel, A. Mavridou, T. Pressburger, M. Bęś, E. Blanco Viñuela, A. Katis, J.-C. Tournier, K. V. Trinh, and B. Fernández Adiego, "Automated verification of programmable logic controller programs against structured natural language requirements," 2023.

[15] Z. Adam, "Formulating requirements with fret for plcverif," Tech. Rep., 2022.

[16] J. Xiong, G. Zhu, Y. Huang, and J. Shi, "A user-friendly verification approach for iec 61131-3 plc programs," *Electronics*, vol. 9, no. 4, p. 572, 2020.

[17] M. Weiß, P. Marks, B. Maschler, D. White, P. Kesseli, and M. Weyrich, "Towards establishing formal verification and inductive code synthesis in the plc domain," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*. IEEE, 2021, pp. 1–8.

[18] M. Niang, B. Riera, A. Philippot, J. Zaytoon, F. Gellot, and R. Coupat, "A methodology for automatic generation, formal verification and implementation of safe plc programs for power supply equipment of the electric lines of railway control systems," *Computers in Industry*, vol. 123, p. 103328, 2020.

[19] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," arXiv preprint arXiv:2202.05218, 2022. [Online]. Available: https://arxiv.org/abs/2202.05218

[20] M. Ebrahimi Salari *et al.*, "Automated test generation for plcs via python translation," in *ICST Workshops*, 2023.

[21] I. D. Lopez-Miguel, B. F. Adiego, M. Salinas, and C. Betz, "Formal verification of plcs as a service: A cern-gsi safety-critical case study," in *NASA Formal Methods Symposium*. Springer, 2025, pp. 227–235.