Bridging Function Block Diagrams and Verified Python Code: A Verification-Aware Approach for Programmable Logic Controllers

Mikael Ebrahimi Salari[1], Cristina Seceleanu[1], Eduard Paul Enoiu[1], Marco Eilers[2], Alessio Bucaioni[1], and Wasif Afzal[1]

[1] Department of Computer Science and Engineering, Mälardalen University, Västerås, Sweden
mikael.salari@outlook.com, cristina.seceleanu, eduard.paul.enoiu, alessio.bucaioni, wasif.afzal[@mdu.se]
http://www.mdu.se
[2] Department of Computer Science, ETH Zurich, Zurich, Switzerland
marco.eilers@inf.ethz.ch
http://www.ethz.ch

Abstract. Translating programmable logic controller (PLC) programs into analyzable artifacts can enable modern testing and verification techniques in industrial settings. PLC Function Block Diagram (FBD) programs differ from traditional software because they execute cyclically over physical inputs and outputs, combine signal flow with stateful and timed blocks, and depend on scan-cycle semantics, environment assumptions, and block-level abstractions. This paper presents a Python-based verification approach that combines *PyLC+*, a translation framework for IEC 61131-3 FBDs, with the *Nagini* verifier. *PyLC+* extracts block networks from PLCopen XML and generates executable Python models that preserve the extracted topology, signal flow, and scan-cycle ordering, while making vendor-specific semantics explicit through reconstruction or abstraction. We apply the approach to 13 industrial program organization units (POUs), where 12 are verified against concrete or partially abstracted FBD-derived models, and one against an abstract requirements-level specification. The results provide evidence that automated extraction, manual semantic completion, and contract-based specification can support deductive verification of realistic POU-level models. The guarantees apply to the stated verification models, not to the full original PLC logic including all vendor-specific and timing semantics. We discuss the required abstractions, a *Nagini* issue exposed by one POU, and missing block semantics that currently limit further automation in *PyLC+*.

Keywords: PLC, Formal Verification, Nagini, Viper, large language models, Industrial Automation.

1 Introduction

Programmable Logic Controllers (PLCs) continue to form the computational backbone of industrial automation systems [1]. Their safety-critical role, particularly in transportation, energy, and process industries, has intensified the need for strong assurance techniques that go beyond software testing. PLC programs, however, differ from traditional software in ways that directly affect verification. PLC programs are typically executed cyclically: each scan reads physical inputs, evaluates control logic in a deterministic order, updates the internal states, and writes outputs to actuators. In Function Block Diagrams (FBDs), this logic is expressed as graphical signal flow between blocks. Verification must therefore take into account scan-cycle semantics, signal propagation, retained states, timers, and data types. Yet, the adoption of formal verification in industry is limited by two long-standing obstacles [2, 3]: the semantic gap between PLC artifacts and verification-oriented representations, and the substantial manual effort required to formulate specifications, harnesses, and environment assumptions.

Recent research has explored translating PLC programs into analyzable models in higher-level languages such as Python or C, thereby enabling the use of modern verification tools [4, 5]. However, these approaches are often demonstrated using simplified examples or rely on handcrafted models that do not capture 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.

In their previous work, Salari et al. introduce *PyLC+* [6], a translation and testing framework that converts Function Block Diagram (FBD) programs, one of the IEC 61131-3 PLC graphical languages expressed as interconnected function blocks, from PLCopen XML into executable Python code, for automated test generation. This work advances *PyLC+* by using Python as a verification-aware bridge between industrial FBD artifacts and deductive verification. *PyLC+* reconstructs the FBD topology and scan-cycle order as executable Python, while *Nagini* [7], a static verifier for Python, checks contracts over the translated model by compiling annotated Python code to the Viper intermediate language and discharging verification conditions through the Viper [8] infrastructure. The main idea is not to verify the PLCopen XML directly, but to make the PLC execution model explicit in Python and then state PLC-specific requirements as *Nagini* contracts. 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, are created semi-automatically: an LLM is used to generate scaffolding and assist with boilerplates, while semantic constraints, invariants, and pre- and post-conditions are refined manually.

FBD-based PLC programs are natural candidates for model checking, especially when the verification target is a finite-state abstraction and the properties

are temporal. Our use of deductive verification is complementary rather than a replacement for model checking. We investigate whether industrial FBD artifacts can be translated into executable, contract-bearing Python models and verified modularly at the POU level. This is useful for data-dependent functional relations, explicit environment assumptions, and reusable contracts for recurring block patterns. A comparative evaluation with PLC-oriented model checking, including temporal and trace-based timing properties, is outside the scope of this paper and remains as future work.

Python is used as a pragmatic verification-aware target rather than as a claim that it is the uniquely best semantic representation of FBDs. It allows us to reuse the existing PyLC translation pipeline, retain an executable model for testing and inspection, and connect the model to Nagini and the Viper verification infrastructure. A translation to a functional language with a dependent-type verification framework could offer different advantages, particularly for purely functional fragments. Such a choice would, however, require a different translation and verification tool chain, as well as separate evidence about its treatment of scan-cycle state, timers, and vendor-specific blocks. Comparing target-language choices is left for future work.

We apply this workflow to a set of 13 real-world industrial Program Organization Units (POUs) from a safety-critical control subsystem. These POUs cover a diverse set of control responsibilities, including redundant-sensor validation, brake supervision, and brake state evaluation. We verify the corresponding POU-level verification targets in Nagini, out of which 12 targets are based on concrete or partially abstracted FBD-derived Python models, while one target is an abstract requirements-level model. The latter POU's failed verification reveals a limitation in the released *Nagini* version's support for tuples larger than six elements. Based on this finding, the tool is extended to handle longer tuples, and the harness then verifies successfully. Our claims concern the correctness of the stated models under their explicit assumptions and abstractions, rather than full semantic equivalence to the original PLC artifacts in all vendor-specific and timing details. The study highlights both the strengths and limitations of translation-based verification: the approach succeeds for most realistic control logic but still requires human judgment for semantic reconstruction and contract definition.

Our contributions are fourfold. First, we clarify the verification gap between FBD programs and traditional software by focusing on scan-cycle execution, signal flow, state, timers, and environmental assumptions. Second, we present a *PyLC+* and *Nagini* workflow that translates PLCopen XML FBD programs into executable Python models and verifies POU-level properties using reusable, POU-specific *Nagini* harnesses. Third, we illustrate how the approach works on industrial POUs by connecting translated FBD logic, block specifications, state abstractions, and POU-specific contracts. Fourth, we evaluate the workflow on 13 industrial POUs, distinguishing concrete FBD-level models, partially abstracted models, and one abstract requirements-level model, while highlighting remaining semantic gaps and tool limitations that constrain the interpretation

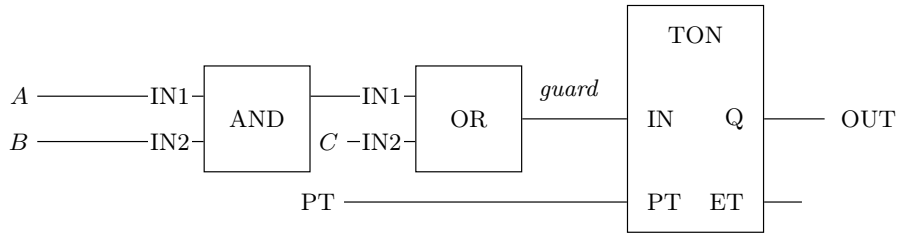

Fig. 1. FBD example with an on-delay timer (TON): $guard = (A \wedge B) \vee C$ and OUT = TON.Q.

of the verification results. The remainder of this work is structured as follows. Section 2 introduces the necessary background on PLC semantics, *PyLC+*, and the *Nagini*/Viper verification framework. Section 3 states the research goal and objectives. Section 4 presents the system overview and verification-aware workflow. Section 5 details the harness architecture, PLC semantics, and verification scope used in the study. Section 6 reports the verification results for 13 industrial POUs. Section 7 evaluates the pipeline through structural and performance metrics. Section 8 discusses the findings and their limitations, and Section 9 surveys related work. Section 10 concludes with directions for future research.

2 Background

PLCs, FBDs, and Scan cycles. The IEC 61131-3 standard [1] defines PLC programming languages ranging from textual languages, such as Structured Text (ST) and Instruction List (IL), to graphical languages, such as Ladder Diagrams and Function Block Diagrams.

Function Block Diagrams (FBD) are graphical, data-flow-oriented programs built by connecting logical functions, arithmetic operations, timers, counters, and vendor components. Each scan cycle propagates values through these networks. At the same time, stateful elements (e.g., RS latches) and *on-delay timer* blocks (e.g., `TON`) maintain state across cycles, making their semantics central to the translation, testing, and verification of such artifacts.

A small FBD example is shown in Fig. 1. It illustrates basic logic blocks (`AND`, `OR`) and a stateful (`TON`) block, which delays a rising input signal by a specified time. During each scan, the Boolean blocks compute $guard = (A \wedge B) \vee C$. The timer receives this signal through `IN` and the preset through `PT`; its output `Q` becomes the POU output.

Many industrial FBD programs rely on the IEC 61131-3 on-delay timer `TON`, whose behavior depends on how it responds 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. In this study, we do not verify millisecond-accurate timer internals. Instead, the harness abstracts the TON output as an explicit Boolean signal indicating that the relevant delay has elapsed.

The verified properties therefore concern how the surrounding POU logic reacts to timer expiration, not whether the timer implementation itself realizes a faithful timing semantics. Detailed timer semantics are left to future work.

The blocks of Fig. 1 appear frequently in industrial control applications and are referenced throughout the paper (e.g., `TON`, `AND`, `OR`, `ADD`).

***Nagini* and *Viper*.** *Nagini* [7] 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 [8]. *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 back-ends, *Silicon* and *Carbon*, attempt to discharge these VCs by employing automated theorem proving and satisfiability modulo theories (SMT) solving using the *Z3* SMT solver [9]. In essence, *Nagini* acts as a front-end that translates Python code and contracts into the Viper intermediate language, while Viper and its back-ends perform the underlying logical reasoning to ensure program correctness.

Illustrative Example. Listing 1.1 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 1.1. A simple Nagini example with a pre- and post-condition using `nagini_contracts` (syntax adjusted for presentation purposes)

```
from nagini_contracts.contracts import *

def increment(x: int) -> int:
    Requires(x >= 0)
    Ensures(Result() >= 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. We use "pure specification function" to mean a side-effect-free Python/-Nagini helper used only to define expected logical behavior in contracts; it is not part of the executable PLC implementation. (ii) *Permission-based* separation logic [10,11] 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*. In this paper, the same contract style is used for PLC-specific obligations: translated FBD outputs must match Boolean specification functions, latches must satisfy their contracts, and POU-level safety outputs must satisfy their stated requirements under explicit environment assumptions. Moreover, SAFE datatypes denote safety-qualified variants of PLC datatypes used in the industrial library. In the Nagini harnesses, their relevant value domains, validity/status conditions,

and range assumptions are represented explicitly as preconditions and helper predicates.

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 [12,13]. A PLC scan reads inputs, evaluates networks in a deterministic topological order, and then updates outputs.

3 Research Goal and Objectives

The goal of this work is to assess how far translation-based modeling, supported by semi-automated harness construction, can enable formal verification of industrial PLC programs. We focus on a practical *PyLC+* and *Nagini* workflow that reconstructs PLC semantics in Python, expresses functional specifications, and verifies safety-relevant POU-level logic.

3.1 Objectives

To realize this goal, we pursue the following objectives:

- **O1 (Verification-Oriented Semantic Completion).** Use PyLC+ to extract FBD topology, connections, typing information, and execution order into Python. Supply executable definitions or explicit abstractions for block behavior that is absent from PLCopen XML or is vendor specific, including SAFE-library blocks, retained state, and timers. This includes implementing or specifying common blocks (e.g., `AND_S`, `OR_S`, `EQ_S`). For timers, the current implementation uses an explicit flag abstraction (e.g., `RS_S`, `TON_S`) and applies manual reconstruction when vendor-specific behavior is not fully defined.
- **O2 (Specification and Harness Construction).** Develop verification harnesses in *Nagini* that express functional contracts, domain assumptions, and safety properties. Harnesses are created in a semi-automated manner: an LLM provides the structure, while semantic details, such as invariants, SAFE range assumptions, environment assumptions, and abstraction boundaries, are refined manually.
- **O3 (Verification of Industrial POUs).** Apply the workflow to a representative set of 13 POUs from an industrial control subsystem. Determine which POUs can be fully reconstructed, which require abstraction, and which stress the current verification tool-chain (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

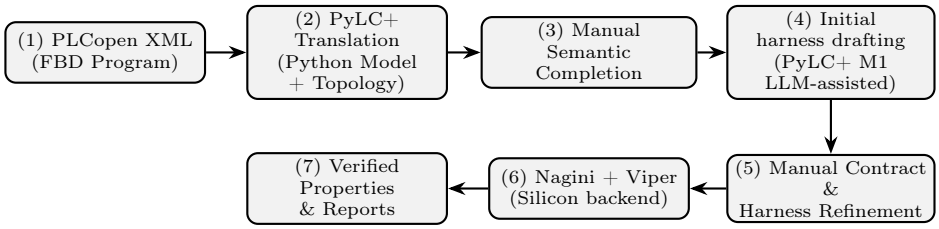

Fig. 2. Verification-aware workflow used in this study. *PyLC+* extracts structure from PLCopen XML; manual semantic completion fills in domain- and vendor-specific behavior; *PyLC+ M1* assists with scaffolding verification harnesses; final contracts are refined manually before verification with *Nagini*.

automation in *PyLC+*, including expanded block semantics and improved support for contract generation.

4 System Overview and Workflow

The approach uses Python to connect industrial FBD programs and deductive verification. *PyLC+* first translates the PLCopen XML representation into a Python model that makes the FBD topology, signal dependencies, and scan-cycle order explicit. *Nagini* contracts are then written on this Python model to express the intended behavior of blocks, networks, and complete POUs. The verifier does not reason about the graphical FBD, but it reasons about a Python artifact whose structure reflects the PLC program. Assumptions are made explicit in the harness.

As illustrated in Fig. 2, the verification-aware pipeline consists of seven steps. The process begins with importing 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 behavior. In step 4, the *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, pre-conditions, post-conditions, 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 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.

4.1 Verification Scope and Semantic-Preservation Assumptions

Let P denote the PLCopen XML representation of a POU. PyLC+ generates a typed Python translation $T(P)$. After any required manual semantic completion or abstraction, the artifact actually verified is denoted by $M(P)$. A POU-specific harness H constrains admissible inputs and prior state and expresses the property φ as Nagini contracts.

A successful Nagini run establishes that $M(P)$, under the preconditions in H, satisfies φ, subject to the soundness of the Nagini/Viper toolchain. Since Nagini does not analyze P directly, transferring φ to the original FBD is conditional on the following assumptions.

A1—Structural extraction. All property-relevant variables, constants, ports, blocks, connections, and network boundaries are extracted correctly. Anonymization and filtering preserve the structure and values relevant to φ.
A2—Execution correspondence. One invocation of the translated `step` operation represents one PLC scan, and the input sampling, block-evaluation order, state updates, and output updates agree with the original POU.
A3—Datatype correspondence. Python values and harness preconditions represent correctly the property-relevant ranges, validity information, conversions, and partial operations of the corresponding IEC 61131-3 and SAFE datatypes. Unmodeled overflow, rounding, floating-point, or vendor-specific behavior is excluded or documented as an abstraction.
A4—Supported-block semantics. Every concretely represented block agrees with its relevant IEC or industrial-library semantics over the domain admitted by the harness, including enabling conditions, priorities, state transitions, and outputs.
A5—Initial and retained state. Initial values and state retained between calls to `step` correspond to the original POU. Any arbitrary or constrained initial state used for verification is stated explicitly in the harness.
A6—Semantic completion and abstraction. Manually reconstructed or abstracted blocks preserve all behaviors relevant to φ, possibly under additional documented assumptions. In particular, timer and vendor-specific block abstractions must be interpreted according to their stated abstraction boundaries.
A7—Environment assumptions. The preconditions imposed on external inputs, validity indicators, previous states, and scan-related parameters hold in the intended operating context.

The proof also depends on the adequacy of the contract: verification shows that $M(P)$ satisfies the stated property, but does not establish that the property is complete or correctly derived from the engineering requirements. Properties inferred primarily from the implementation should therefore be interpreted as consistency properties. The current work does not provide a machine-checked simulation, refinement, or semantic-equivalence proof between PLCopen/FBD execution and the Python models. The results are therefore conditional guarantees for the classified verification models. A formal proof of translation correctness, or translation validation for each generated POU, remains to be tackled in future work.

Table 1. Interpretation of verification results by model class.

Model class	Model correspondence	Meaning of successful verification
Concrete FBD-derived	The property-relevant FBD structure and supported block behaviour are represented directly in Python.	Conditional evidence for the original POU, provided A1–A5 and A7 hold. A6 additionally applies wherever manual semantic completion remains.
Partly-abstracted FBD	The FBD structure remains recognisable, but some timing, datatype, block, or state behaviour is abstracted.	A guarantee for the partly abstracted model. Transfer to the original POU requires A6 and does not cover behaviour removed by the abstraction.
Abstract requirements-level	The model expresses a requirements-level function or state machine without a one-to-one mapping to the FBD networks.	A proof about the abstract model and its contracts. No claim about the original FBD follows without a separate refinement or conformance argument.

We distinguish three model classes according to the correspondence between $M(P)$ and the original FBD, as shown in Table 1. Accordingly, *verified POU* means that Nagini discharged the stated obligations for the reported model and harness. The strength of the source-level guarantee depends on the model class and on the assumptions above.

5 Method

The automatic translation of PLCopen XML FBD programs into Python using *PyLC+* has already been evaluated in earlier work [6]. This section therefore focuses on what is *verified* after translation, which PLC semantics are reconstructed in the Python model, which semantics are abstracted in the harness, and how *Nagini* contracts are used to check POU-level requirements. We first describe the harness architecture, 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.

5.1 Harness Architecture

A verification harness is a small, verification-specific wrapper around a translated POU. It instantiates or invokes the POU model, constrains admissible inputs and relevant prior state through assumptions and preconditions, and states the property to be checked through contracts. The harness is not an independent reimplementation of the POU: it combines the translated POU code with shared specification helpers, explicit state abstractions, and POU-specific requirements. The harness-based verification architecture is shown in Fig. 3. 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 creation and verification:

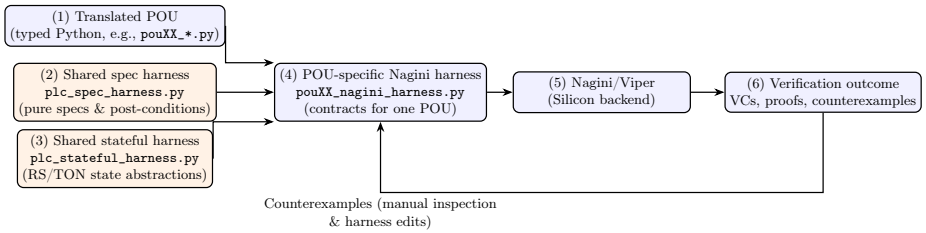

Fig. 3. Harness-based architecture actually used in this work. Each translated POU module is combined with a shared spec, stateful harnesses, and a POU-specific *Nagini* harness, with interactive assistance from an LLM (*PyLC+ M1*). *Nagini* (via Viper's Silicon backend) discharges verification conditions and returns counterexamples for manual inspection.

- **Translated POU (step 1).** For each industrial POU, *PyLC+* translates the PLCopen XML description into a Python module that preserves the FBD structure, scan-cycle order, and stateful block interfaces.
- **Shared spec harness (step 2).** This module defines pure specification functions and post-conditions for block patterns and POU-level outputs. The functions are written in *Nagini*'s contract language (`Requires`, `Ensures`) and are reused across multiple POUs.
- **Shared stateful harness (step 3).** This module contains small explicit classes (plain Python classes) and helper functions that model the abstract state of RS latches and on-delay timers. Timer-dependent POUs use either the T1 Boolean expiration abstraction or the T2 discrete scan-counter abstraction defined in Section 5.2. The shared contracts are sufficient for the POU-level properties that we verify here, but do not model millisecond-accurate vendor 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 Python via preconditions, post-conditions, 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 iterations on the harnesses and, in one case, on the verifier itself. In particular, the abstract requirements-level harness for `POU_05` initially triggered an internal *Nagini* error due to a 7-tuple in a pure specification function, revealing a bug in *Nagini*'s handling of tuples longer than 6 elements. The *Nagini* developers fixed this bug in the master branch, and with the patched version `POU_05`, it verifies as the other POUs do. When verification failed during development,

we manually inspected the counterexample traces and adjusted the harness (e.g., specifications or preconditions).

Importantly, the current artifact does not implement an automated feedback loop from Nagini to the harness. Counterexamples are interpreted and the harnesses are refined manually (with occasional LLM assistance for code editing), for the reasons discussed in Section 5.4. The source artifacts are maintained under version control, but independent reproduction is limited because the repository cannot be made publicly available, due to restrictions on the dissemination of the industrial artifacts.

5.2 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. We consider three categories of behavior: combinational logic, RS latches, and on-delay timers (TONs).

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.

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 data class and a single-step update function:

```
from nagini_contracts.contracts import *

class RsState:
    def __init__(self, q: bool) -> None:
        self.q = q

def rs_step(old_state: RsState, i_set: bool, i_reset: bool)
    -> RsState:
    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))
```

```
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 post-conditions 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.

On-Delay Timer (TON) Abstraction. The industrial POUs include TON blocks whose concrete behavior depends on cycle time and a preset threshold PT. The verification models do not attempt to reproduce a millisecond-accurate vendor implementation. Instead, timer treatment is POU-specific, and the current artifact uses the following two abstractions.

T1—Boolean expiration abstraction. The output `TON.Q` is represented by an explicit Boolean input, denoted by `timer_expired` or `timer_q`, indicating whether the timer has expired. The harness does not compute this value from an internal clock or elapsed-time state. Nagini verifies only that the surrounding POU logic reacts correctly to the two possible timer-output values (e.g., gating brake fault signals after a minimum activation period). Unless further constrained by a POU-specific precondition, `timer_expired` is unconstrained at the verification boundary.

For example, the following verification harness exposes the abstract timer output:

```
# T1: expiration supplied at the verification boundary
from nagini_contracts.contracts import *

def ton_q(timer_expired: bool) -> bool:
    Requires(True)
    Ensures(Result() == timer_expired)
    return timer_expired
```

T2—Discrete scan-counter abstraction. For POUs requiring threshold-related reasoning (e.g., `POU_8`), elapsed time is represented by a non-negative discrete counter that is updated once per modeled PLC scan. Under an assumed fixed scan duration, the timer preset is represented as a corresponding number of scan steps. The harness may then verify properties such as counter monotonicity while the timer input remains active, reset behavior when the input becomes inactive, and that `timer_q` cannot become true before the abstract threshold is reached.

The T2 model is a scan-level abstraction rather than a physical-time model. Its interpretation relies on the assumptions that one invocation of `step` represents one PLC scan, that the scan duration used to derive the counter threshold

is fixed for the considered execution, and that the counter update and reset rules correspond to the timer behavior relevant to the verified property.

The following simplified, generic pattern illustrates the obligations, yet concrete names differ among POU harnesses:

```
# T2: one abstract update per PLC scan
def timer_step(old_count: int, active: bool,
               PT: int) -> TimerState:
    Requires(old_count >= 0)
    Requires(PT > 0)
    Ensures(Implies(not active, Result().count == 0))
    Ensures(Implies(active,
            Result().count >= old_count))
    Ensures(Implies(Result().q,
            Result().count >= PT))
    ...
```

Neither T1 nor T2 establishes millisecond-accurate semantic equivalence to a concrete vendor TON implementation. T1 verifies only the reaction of the FBD surrounding logic to timer expiration. T2 additionally verifies selected scan-level threshold and state properties, but abstracts scheduling jitter, execution-time variation, sub-scan input changes, and vendor-specific timing details. Each POU result must therefore be interpreted together with its reported timer-abstraction class.

5.3 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 using 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 post-conditions 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.

The harnesses invoke the translated POU implementations; they do not replace them with an independently executable replica of the complete FBD network. Their specification functions express selected expected output relations, safety conditions, and admissible environment assumptions. Some contracts necessarily reflect implementation-level knowledge, such as the assumed semantics

of reconstructed blocks and the state abstractions used for RS and TON behavior. Therefore, successful verification should be interpreted as consistency between the translated POU model and the stated requirements under these assumptions, rather than as an independent fault-detection experiment over the original PLCopen XML. During development, unsuccessful proof attempts were used to refine models, assumptions, and contracts; the POU 05 case also exposed a limitation in Nagini. In summary, the current replication uses handwritten (and sometimes LLM-assisted) contracts at the block and POU levels, without the fully automated property extraction and POU-composition algorithms originally envisioned. We treat those algorithms as future work rather than part of the present contribution.

5.4 LLM-Assisted Harness Authoring

PyLC+ M1 is the Qwen-based interactive model introduced in our earlier work [6]. Although the original design considered more automated harness synthesis, in this study we use the *PyLC+ M1* model in a more modest role: as a *coding assistant* for human-authored harnesses.

Concretely, the workflow for each POU is:

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 request a draft harness skeleton (file *pouXX_ nagini_ harness.py*).
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 are manually reviewed, edited, and stored in version-controlled source files. The reported results do not depend on hidden LLM state, model fine-tuning, or an automated refinement service. Counterexamples are not automatically returned to the LLM because the traces contain verifier-specific heap and permission information and may admit several semantically different repairs. Human inspection is therefore required to classify whether a failure indicates a missing environment assumption, an incorrect or overly strong contract, an unsupported construct, or an implementation-model mismatch. Automated counterexample-guided harness repair is not evaluated in this study.

Harness review procedure. Harnesses have been authored by the first and the fourth authors familiar with the translated PLC code and Nagini, and then cross-checked by additional authors against the translated Python, extracted FBD data, and informal industrial requirements. The properties encoded in the POU-specific harnesses originate from several types of sources: system or safety requirements, interface documentation, industrial block-library documentation,

engineer-provided natural-language requirements, and behavior inferred by inspecting the FBD and its translated Python model.

For each POU, we check that the final contract refers only to documented inputs, outputs, SAFE-type assumptions, and explicitly listed abstractions. The counterexamples produced during development have been inspected manually and classified as either missing assumptions, overly strong specifications, or genuine harness errors.

The authors retain the translated models, curated harnesses, and verification logs in a version-controlled repository. However, restrictions on the dissemination of the industrial artifacts prevent the public release of this repository. Consequently, the reported workflow is internally repeatable from the artifacts, but full independent reproduction is not currently possible.

5.5 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.

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)

def check_SMgBrIsl(x: int, v: bool, mio: bool, C: int) ->
    bool:
    Requires(x >= 0)
    Ensures(Result() == spec_SMgBrIsl(x, v, mio, C))
    return translated_SMgBrIsl(x, v, mio, C)
```

Here, `spec_SMgBrIsl` states the expected Boolean relation (derived from the requirement), whereas `translated_SMgBrIsl` denotes the corresponding implementation generated from, or invoked through, the translated FBD. Nagini therefore checks the translated implementation against a separate specification. This pattern appears, for example, in the brake isolation logic and in simpler sanity checks.

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.

TON harness The POU-specific harnesses instantiate one of the two timer abstractions defined in Section 5.2. Most of the POUs use the T1 pattern, where the harness receives an explicit Boolean timer-expiration signal and verifies the

behavior of the surrounding POU logic for both its false and true values. It does not verify elapsed time or the timer preset.

Some POUs, e.g., `POU_8`, use the T2 pattern, where the harness maintains a discrete elapsed-scan counter and states contracts relating the timer input, the previous counter value, the updated counter, the abstract preset threshold, and `timer_q`. This supports verification of selected reset, monotonicity, and not-before-threshold properties under the fixed-scan-duration assumption. The choice between T1 and T2 is made per POU and is part of the documented abstraction boundary. Neither pattern verifies the complete millisecond-accurate or vendor-specific implementation of the TON block.

Overall, the Methods in this paper therefore correspond exactly to what is implemented: 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.

6 Verification Results

Following the verification-aware pipeline in Fig. 2, we applied *Nagini* to the translated Python versions of the industrial POUs and checked properties at the block, network, and POU levels. Functional obligations focus on Boolean and comparison logic, reset-dominant latch behavior, and the agreement between outputs and functions. Timer-related obligations use the POU-specific abstraction defined in Section 5.2. T1 models verify only the surrounding logic's response to a Boolean expiration signal (the harness treats the timer output as an explicit signal indicating that the relevant delay has elapsed), whereas T2 models also check selected scan-counter properties, such as reset, monotonicity, and not-before-threshold conditions. Neither abstraction proves millisecond-accurate TON behavior. T2 results additionally assume one modeled update per PLC scan and a valid fixed scan duration and threshold.

Using the Silicon backend, we obtain proofs for all **13** industrial POUs from the industrial-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 data types.

POU05 and a Nagini issue. In the original experiments, the `POU_05` harness did not verify because *Nagini*'s backend reported an issue when translating a pure specification function returning a 7-element tuple. We traced it to an unsupported tuple size that triggered an internal error; this has since been corrected on the current master branch. With the patched version, the `POU_05` harness

verifies successfully. We therefore include `POU_05` among the verified POUs, noting that verification is at the level of its abstract requirements model and that the harness helped uncover and resolve this tooling issue.

Our observations suggest that POUs involving retained state or timer abstractions require more detailed assumptions and longer verification run times. For some timing and arithmetic-heavy POUs, we used abstract harnesses and verified high-level properties over an abstract model rather than the complete low-level FBD logic. Therefore, the verification claim should be read together with the model classification reported for each POU. All reported successful runs mean that Nagini discharged the stated proof obligations for the corresponding verification model and harness. They do not show that the original PLCopen XML contains no logical defects, because the evidence is conditional on the translation assumptions in Section 4.1, the selected requirements, and the abstractions used for unavailable vendor-specific or timing semantics. Likewise, a failed proof attempt during development may reveal an implementation-model mismatch, an insufficient assumption, or an incorrect or incomplete contract. Because no systematic implementation-level mutation experiment was performed, these development outcomes were not used to estimate defect-detection effectiveness. We therefore report verification success as model-level evidence, together with the model class and abstractions used.

7 Evaluation

We now evaluate the verification-aware pipeline on the 13 industrial control POUs selected for the case study. 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 are used during harness development.

7.1 Programs and Levels

The case-study system contains multiple anonymized 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 focus 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 modeling. 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 behavior, and dynamically sized arrays updated across scan cycles. POUs relying on such features are therefore excluded. After applying these criteria, **13** POUs remain and are therefore verified: twelve under concrete or partially abstracted FBD models,

and POU_05 under an abstract requirements-level specification. POU_05 has also exposed a tuple-length limitation in *Nagini*, which is resolved in the patched version used in this study.

For each POU, we consider three verification levels within the translated artifact: individual block, FBD network, and complete POU. Here, *level* denotes verification granularity and should not be confused with the IEC 61131-3 POU categories Program, Function Block, and Function.

Block level. We check 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 behavior used in the harness cannot raise its output before the preset delay PT_TRIP_MS.

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), the harness states the documented relation between flags, validity bits, and abstract timer outputs. Likewise, in the plausibility check POU (POU_13), we check 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.

POU level. At the POU level, each POU-specific harness defines one or more top-level check functions whose contracts express industrial requirements over the translated or abstracted Python model. These contracts are checked under the scan and step semantics and assumptions encoded in the harness. The current artifact verifies POU-level properties within the modeled abstraction, subject to explicit assumptions about retained state and abstract timer signals.

7.2 Role of LLM Assistance in Harness Construction

The harnesses and contracts are written in Python with *Nagini* annotations, using the *PyLC+ M1* interactive LLM assistant. For each POU, we use *PyLC+* to generate a typed Python class representing the translated FBD structure and supported block semantics. Contracts and invariants are then drafted with LLM support, based on manually written natural-language descriptions, and manually refined to account for SAFE datatypes, stateful latch behavior, and the timer abstractions. We refine the annotations by hand to satisfy *Nagini*'s restrictions and iteratively run *Nagini*/Silicon, using errors and counterexamples to adjust the code or specifications until verification succeeds. The result is an executable set of *Nagini*-compatible harnesses that capture the intended FBD logic (or a documented abstraction), produced via human-in-the-loop iteration rather than a fully automatic pipeline.

Table 2. Summary of industrial POUs and *Nagini* verification results. Times are taken from the actual `spec_properties_silicon.txt` logs. `POU_05` is modeled 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

7.3 Evaluation Summary

For each POU that we successfully verify, we record the following indicators: (i) #**Blocks**, the total number of FBD blocks in the translated POU (from the extracted data files), (ii) #**Stateful**, the number of stateful blocks (RS latches `RS_S` and on-delay timers `TON_S`), (iii) **Model**, whether the harness models the complete FBD diagram (*concrete FBD*), abstracts some aspects while preserving overall behavior (*partly abstracted*), or treats the POU as a pure requirements-level function without a one-to-one FBD mapping (*abstract only*), (iv) **Status**, whether *Nagini* produced a full VC proof (*verified*) or reported a tool error (*no VC proof*), and (v) **Time (s)**, the wall-clock time for the *Nagini+Silicon* run, as reported in the `spec_properties_silicon.txt` files under `results/POU*/`.

Table 2 summarizes these metrics. For all 13 POUs that we 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**. 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.

Across the verified POUs, we cover a range of control-logic patterns and modeling aspects: the 2oo2 voters/adapters (`POU_01`, `POU_02`, `POU_03`, `POU_12`) are small to medium (10 to 23 blocks), purely Boolean and comparison, stateless, and verify quickly (22s to 26s) under concrete FBD semantics; the mapping and conversion POUs (`POU_04`, `POU_11`) include a larger stateless mapper (`POU_04`, 50 blocks, 21s) and a conversion unit (`POU_11`) whose harness introduces a documented, partly-abstracted total division to avoid partiality. Redundancy, supervision, and plausibility checks (`POU_06`, `POU_07`, `POU_08`, `POU_09`, `POU_13`) span 3 to 28 blocks with 0 to 3 stateful elements and verify in 23s to 27s; notably,

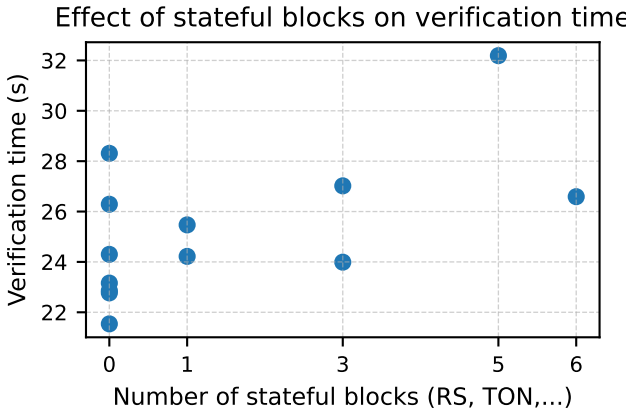

Fig. 4. Effect of stateful blocks (e.g., RS, TON) on verification time. Each point corresponds to one POU.

`POU_09` abstracts timer and latch details into an instantaneous validity predicate plus a per-cycle counter, so the proved properties are high-level validity and counter updates. `POU_10` (park-brake state encoding) has the most stateful blocks among the verified set (5) and is proved under an abstract state-machine model, making it the slowest case (32s). Finally, `POU_05` (hold-brake control) is large (58 blocks, six stateful) and is verified against an abstract requirements-level Python model; an initial *Nagini* tuple-length limitation has caused a tool error, which is fixed, and the patched verifier discharges the harness in 26s, with the caveat that the proof targets the abstract model rather than the complete low-level FBD connections.

7.4 Verification Time and Structural Metrics

According to Table 2, once a POU has a stable harness that respects *Nagini*'s language constraints, full deductive verification completes in a matter of tens of seconds. Figure 4 separates this behavior by the number of stateful blocks (RS latches and *TON* timers) per POU, using counts extracted from the corresponding extracted-data files (denoted *_ *extracted*_ *data.py*).

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.

Figure 5 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,

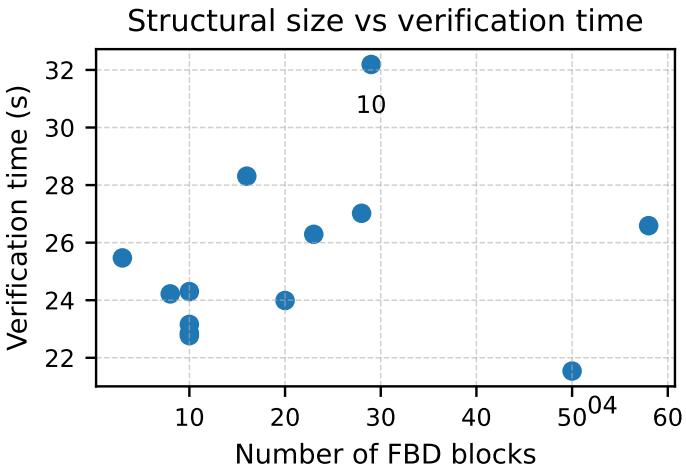

Fig. 5. Structural size (number of FBD blocks) versus verification time. Each point corresponds to one verified POU; the largest combinational POU (`POU_04`) and the slowest POU (`POU_10`) are highlighted.

and therefore shows the largest time (32.19 s) despite not being structurally extreme.

Taken together, these results suggest that verification time is driven more by the complexity of retained state (e.g., RS and `TON` blocks), and timer abstractions, such as RS latches, abstract timer signals, and higher-level state machines, than by the raw number of combinational gates. These observations are descriptive rather than statistical: with only 13 POUs and heterogeneous abstractions, we do not claim a general runtime model. The data are nevertheless consistent with our manual experience that proof effort is influenced more by the kind of state and abstraction used than by raw block count alone. They also indicate that, once a stable harness is available, verification completes within tens of seconds for the POU-level targets considered in this study.

Observed Trends. POUs with stateful blocks (RS and `TON`) verify slightly slower on average (about 26s) than purely combinational ones (about 24s), reflecting the extra invariants needed for latches and timers. For small and medium POUs (up to ∼30 blocks), block count alone is a weak predictor of runtime, with most cases clustering around 22 to 28s and `POU_10` as an outlier due to its richer abstract state machine. Several POUs (`POU_06`–`POU_11`) rely on explicitly documented, partly abstracted harnesses, so the proofs should be read as guarantees for the abstract model rather than FBD-level equivalence to the original program. Overall, within *Nagini*'s current limitations, the experiments show that *PyLC+* enables end-to-end deductive verification in tens of seconds per POU while making abstraction boundaries explicit.

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

We now illustrate the verification-aware workflow of Fig. 3 on one representative anonymized POU, denoted `POU_08`. In the original industrial controller, this POU supervises a braking-related function, but here all identifiers and signal names are anonymized (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. `POU_8` uses the T2 discrete scan-counter abstraction (introduced in Section 5.2). Thus, the timer-related proof obligations in this example concern a scan-level counter and threshold, rather than the T1 Boolean-expiration pattern used by other POUs. The result remains conditional on the fixed-scan-duration and counter-correspondence assumptions stated in Section 5.2.

(1) Translated POU module. Starting from the PLCopen XML, *PyLC+* generates a typed, executable Python translation of the POU, stored as the file *pou08_translated.py*. 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 two `TON`-style timers). 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.

(2) Shared specification harness. The second component of Fig. 3 is the shared specification module, provided as the file *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*, which takes the parameters *raw_code* and *valid_flag* and describes the expected isolation behavior; and (ii) a function *spec_fault*, with parameters *raw_code*, *valid_flag*, and *iso_cmd*, which defines when a fault output must be raised. These functions serve as logical ground truth for the post-conditions in the POU-specific harness.

(3) Shared stateful harness. POUs such as `POU_08` contain stateful behaviors (RS latch, on-delay timer), so the shared stateful harness, provided as the file *plc_stateful_harness.py*, defines 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.

(4) POU-specific Nagini harness. The POU-specific harness, provided as the file *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, normalized integer codes, admissible preset values); (ii) forwards the inputs to the translated POU's `step(...)` method; and (iii) attaches `Ensures` post-conditions checking that the concrete outputs of the translated model match the corresponding pure specification functions and stateful invariants. For example, one post-condition states that the output *iso_out* returned by the implementation must equal the value computed by *spec_iso_status*, given inputs *raw_code* and *valid_flag*, and another ensures that *timer_q* never becomes true before the threshold *PT_ISO*.

To illustrate the verifier's feedback produced by a mutated contract, we deliberately negate one postcondition. This example modifies the specification, not the translated POU implementation, and is not part of the quantitative evaluation.

Listing 1.2. 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(...)
```

This example shows that Nagini rejects a translated model and contract that are logically inconsistent. Because the modification is made to the post-condition rather than to the translated POU, it should not be interpreted as evidence that the workflow detects implementation defects or latent defects in the original PLCopen artifact. Evaluating such sensitivity would require separate implementation-level mutation experiments, which are outside the scope of this study.

(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 behavior:** 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 2 for `POU_08`.

(6) Verification outcome and feedback. Nagini returns either a successful verdict or a counterexample. For the final version of `POU_08`, no counterexamples are produced. During development, unsuccessful verification attempts exposed, for example, (i) missing heap permissions when the post-condition accessed an output field prematurely, and (ii) insufficiently specified timer invariants. 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 that we have 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.

This study evaluates proof feasibility and the interpretation of successful verification for the stated models. It does not evaluate implementation-fault sensitivity or claim that the workflow can detect latent defects in the original PLCopen artifacts.

8 Discussion and Threats to Validity

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

Verification Outcomes. All 13 industrial POUs in the case study were successfully verified using *Nagini*. Twelve POUs were verified against concrete or partially abstracted FBD models, while one POU (`POU_05`) was verified against an abstract requirements-level model. The `POU_05` harness initially exposed a *Nagini* limitation related to tuple size in pure specifications; the developers fixed this issue, after which all verification conditions discharged successfully. The verified POUs span a representative range of industrial control logic, including voting, supervision, plausibility checks, and state encoding, and all completed within tens of seconds (21–32 s), indicating practical feasibility for CI-scale verification.

Patterns and Limits of the Verifier. Verification cost is dominated by stateful behavior rather than structural size. POUs containing RS latches or `TON` timers required additional invariants and verified slightly slower than purely combinational designs, while large but stateless POUs verified quickly. In contrast, moderately sized POUs with richer abstract state machines exhibited the highest run times. In practice, selective abstraction is essential: fully concrete FBD-to-Python modeling is often unnecessary or incompatible with *Nagini*'s restrictions, and small, explicitly documented abstractions (e.g., symbolic timers and higher-level states) were sufficient to preserve relevant behavior while enabling verification.

Toolchain Issue. `POU_05` is a state-heavy POU verified against an abstract requirements-level model. Its harness initially failed due to a *Nagini* front-end limitation when translating pure specification functions that return tuples with more than 6 elements, resulting in an internal error. After this issue has been identified and fixed by the Nagini developers, the `POU_05` harness verified successfully (26 s), illustrating both the feasibility of verifying realistic PLC-derived models and the value of industrial case studies for improving the verification tool chain.

Counterexamples and Harness Development. Counterexamples have been used only during harness construction; no final POU has produced counterexamples. During development, traces expose missing invariants for RS latch reset dominance, signal transition handling, and partially defined numeric operations, which have been addressed through abstraction and contract refinement. Deliberate specification faults (e.g., negated post-conditions) reliably produce counterexamples, demonstrating how *Nagini* traces guide the correction of erroneous or overly strong contracts. In Section 7.5, we illustrate the form of verifier feedback when a contract conflicts with the modeled implementation. Since this modification affects the specification rather than the translated POU, it does not demonstrate sensitivity to implementation faults. No implementation-level mutation study is conducted, and the present evaluation makes no claim about detecting latent defects in the original PLCopen artifacts.

Implications. The results show that translating FBDs to Python with *PyLC+* enables practical deductive verification of realistic industrial programs, provided that semantic gaps and abstraction boundaries are made explicit. All evaluated POUs were verified within tens of seconds using *Nagini* against their concrete, partly abstracted or abstract requirements-level models, making future CI integration feasible for similarly scoped POU-level checks.

Threats to Validity. Threats to validity mainly arise from choices of abstraction when translating IEC 61131-3 semantics to Python. While *PyLC+* preserves network structure, scan-cycle order, and state updates, vendor-specific timer behavior, precise timing, and some arithmetic details are abstracted. External validity is limited by the scope of the case study (13 POUs from a single safety-critical subsystem). Tool chain validity partly depends on a patched version of

Nagini for larger tuples. Although LLM assistance has been used for scaffolding, all contracts are manually reviewed and validated through deductive verification, reducing but not eliminating specification bias. Artifact availability constitutes an additional threat to reproducibility: although the models, harnesses, and verification logs are retained under version control, restrictions on the dissemination of the industrial artifacts prevent their public release and therefore limit independent reproduction of the experiments. The study also does not include an implementation-level mutation analysis. Consequently, it provides no empirical estimate of how frequently the selected contracts would detect erroneous translations or latent defects in industrial POUs.

9 Related Work

Research on PLC verification includes model checking, formal semantics, translation analysis, and hybrid testing–verification workflows. Most approaches translate IEC 61131-3 programs into transition-system models (e.g., SMV or timed automata) for model checking, from early IL translations [14] to industrial-scale studies such as CERN systems [15], with surveys highlighting recurring challenges such as scan-cycle semantics and timers [3]. In parallel, the semantics of IEC 61131-3 have been formalized using Coq [16] and executable frameworks such as K [17]. Other work leverages PLCopen XML for requirement-based verification [18, 19] or reduces specification effort via property and invariant mining [20].

PyLC+ differs by avoiding a separate automaton or intermediate formalism: PLCopen XML is translated directly into executable Python, and deductive verification is performed on the same artifact using *Nagini*. PLC-specific semantics (e.g., reset-dominant latches and timer abstractions) are embedded as contracts, improving traceability and reducing the semantic gap. The approach aligns with hybrid industrial verification workflows [21, 22] and verification-as-a-service efforts [23], and extends earlier PLC-to-Python testing work [24, 25] by unifying execution and proof in a single translated artifact.

10 Conclusions

This study shows that PLCopen XML programs translated by *PyLC+* can be verified deductively using *Nagini* when PLC-specific semantics, environment assumptions, and abstraction boundaries are made explicit in Python harnesses. Thirteen industrial POUs were verified against their stated models: concrete FBD translations, partly abstracted FBD models, or, in one case, an abstract requirements-level specification. The results show that translation-based verification is practical for realistic POU-level industrial logic, with all verification runs completing within tens of seconds once stable harnesses are available. At the same time, the study shows that full automation remains limited by vendor-specific block semantics, timer modeling, manual contract design, and verifier restrictions.

Future work will focus on automating contract and invariant generation from PLCopen metadata, extending support to additional IEC languages and vendor-specific blocks, and improving scalability through modular and incremental verification. Deeper integration of testing and verification, as well as improved counterexample mapping back to PLC artifacts, remains an important direction.

Acknowledgments. This work received funding from the MATISSE project, an EU-funded initiative under Horizon Europe GA no. 101056674, as well as from Mälardalen University, via the TSS-INSID project funded through the Trusted Smart Systems initiative, which supported this research.

References

1. M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010, vol. 166.
2. D. Darvas, I. Majzik, and E. Blanco Viñuela, "Formal verification of safety PLC based control software," in *International Conference on Integrated Formal Methods*. Springer, 2016, pp. 508–522.
3. 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.
4. B. Han, C. Li, H. Deng, G. Liu, and Z. Zheng, "Domain-specific translation tool from structured text to C source code with code readability enhancement in programmable logic controllers," *Concurrency and Computation: Practice and Experience*, vol. 36, no. 15, p. e8100, 2024.
5. J. Li, A. Qeriqi, M. Steffen, and I. C. Yu, "Automatic translation from FBD-PLC-programs to NuSMV for model checking safety-critical control systems," in *Norsk IKT-konferanse for forskning og utdanning*, 2016.
6. 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.
7. M. Eilers and P. Müller, "Nagini: A Static Verifier for Python," in *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, 2018.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.

13. 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.
14. 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.
15. 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.
16. J. O. Blech and S. O. Biha, "On formal reasoning on the semantics of PLC using Coq," *arXiv preprint arXiv:1301.3047*, 2013.
17. 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.
18. 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," National Aeronautics and Space Administration (NASA), United States, Tech. Rep. NASA/TM–20230003752, 2023.
19. Z. Ádám, "Formulating Requirements with FRET for PLCVerif," European Organization for Nuclear Research (CERN), Geneva, Switzerland, Tech. Rep. CERN-STUDENTS-Note-2022-107, 2022.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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
25. M. Ebrahimi 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 *30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 431–440.