Alma Mater Studiorum University of Bologna

DEPARTMENT OF ELECTRICAL, ELECTRONIC AND INFORMATION ENGINEERING
Master degree in Automation Engineering
AUTOMATION SOFTWARE AND DESIGN PATTERNS

MODEL-BASED AUTOMATED GENERATION OF SIEMENS PLC PROJECTS USING TIA PORTAL OPENNESS AND FACTORY I/O

Thesis of:
ANGELICA MARCONE

Supervisor:
EMINENT PROF. GIANLUCA PALLI

Co-supervisors:
GIUSEPPE CANNIZZARO
AHMED MOHSEN MOHAMED FATHY

Session of March

Academic Year 2024/2025

Abstract

In Today's Industry context, the demand of PLC-based automation systems is continuously increasing, leading to a growing need to automate and accelerate the creation and update of PLC programs, while reducing human error. In many engineering workflows, PLC development still relies on manual and point-and-click activities, which are time-consuming, repetitive, and prone to inconsistencies. Modern automation systems therefore are required to support modularity, reusability, and frequent reconfiguration, which are key aspects in the development and management of reconfigurable manufacturing systems.

This thesis presents the development of an automatic tool that supports the design, creation, and update of industrial automation projects in *Siemens Totally Integrated Automation (TIA) Portal*. The main objective is to reduce engineering effort and likelihood of human errors by automating repetitive and error-prone tasks performed during PLC project development, including hardware configuration, I/O addressing, and the generation of software structures such as Function Blocks and Data Blocks.

The proposed approach is based on a specification-driven workflow, where the characteristics of the industrial process are described through user-editable configuration files, such as Excel or XML documents. These specifications are interpreted by a C# application exploiting Siemens *TIA Portal Openness*, enabling the automatic generation and update of PLC projects in a consistent and reproducible manner.

The thesis is developed through two parallel activities. The first focuses on the implementation of the C# automation tool for TIA Portal, covering project creation, hardware setup, rack configuration, and control software generation. The second activity concerns the modelling of the industrial process in Factory I/O, which is used as a digital twin environment to define sensors and actuators and ensure a consistent mapping between the simulated process and the PLC project.

The case study results suggest that the proposed approach can improve consistency, reusability, and maintainability of PLC projects, while significantly reducing manual engineering effort and commissioning risks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Industrial context

In recent years, the evolution of automation systems has been one of the key enablers involved in the Industry 4.0 paradigm, which describes the transition toward highly digitalised, interconnected and intelligent industrial systems. This shift has significantly increased the complexity of PLC-based control applications, leading to higher engineering effort and tighter development timelines. According to an official report published by the European Parliament [5], future production processes and automation systems will increasingly rely on virtual design and commissioning within integrated and collaborative development frameworks, reducing the need for physical prototypes. As a consequence of the increasing connectivity, data integration, modularity and shorter lifecycle, promoted by the Industry 4.0 paradigm, industrial automation systems have evolved from isolated control solutions toward large-scale, distributed architectures integrated with higher-level infrastructures. This evolution has resulted in a growing number of interfaces, signals and software modules, increasing the engineering effort required for PLC-based control systems. At the same time, this change in production systems has enabled increased productivity and flexibility, but also increased complexity in industrial processes, from technological and engineering perspective. In particular, this complexity is reflected in the increasing scale of I/O configurations, highly modular software architectures, frequent system updates and stricter requirements for traceability throughout the engineering lifecycle.
Modern automation systems must coordinate heterogeneous devices, manage complex software architectures, and support frequent modifications to control logic and system configurations of different components in response to changing operational requirements. These activities typically involve hardware configuration, I/O addressing, tag consistency management and the reuse of software FBs across dif-

ferent system components. In this context, traditional manual engineering, based on point-and-click configuration workflows, becomes inadequate in terms of commissioning time and susceptibility to human errors. The current industrial practice requires faster processes and system reconfiguration.
To meet these goals, the digital transformation of automation engineering has led to the increasing adoption of model-based design and specification-driven workflows, supported by the integration of digital engineering tools. These approaches contribute to reducing engineering effort, improving consistency and enhancing the reliability and maintainability of complex industrial automation systems. In this context, this thesis focuses on automating PLC project creation and evolution within the Siemens TIA Portal environment.

PLC programming challenges

The qualification of industrial control systems involves extensive verification and validation activities to ensure compliance with functional, safety and reliability standards, making it a costly and time consuming process. Consequently, the reuse of qualified hardware and pre-certified software becomes increasingly important, enabling engineering efforts to be focused on application-level software. This led to the adoption of (PLCs), which enable application programming within standardised IEC 61131-3 environments, simplifying system certification [6].
In traditional industrial automation workflows, PLC projects are often created manually within software as TIA Portal, requiring engineers to set by hand hardware configuration, I/O addressing, tag table and software structure. Although this approach works well in principle, it introduces significant inefficiencies in the commissioning timelines of modern, large-scale automation systems, mainly due to frequent configuration changes, multiple system variants, extended I/O sets and multi-device architectures. Consequently, engineers increasingly adopt automation and model-based approaches to reduce repetitive low-level work. This shift addresses several limitations of manual programming, including limited reusability, higher error rates and the high effort required to perform updates, adaptations and system reconfiguration, especially in complex industrial systems.

Problem identification

In environments where flexibility and rapid change are increasingly demanded, the traditional workflows are no longer able to meet the demanded levels of performance, in terms of engineering throughput, configuration consistency and traceability. In order to address emerging industrial requirements, several new approaches

have been proposed and adopted. However, existing solutions typically focus on specific phases of the engineering process and do not fully automate PLC project generation and update workflows in an end-to-end manner within Siemens TIA Portal

For example, as explained in an industrial process programming [7], the combination of model-based design techniques and external Python packages with Siemens TIA Portal, allows engineers to design and validate control strategies for an industrial thermal process as an alternative to traditional manual workflow. The proposed solution improves controller design, validation and commissioning by exploiting high-level modelling and simulation in Python. However, it mainly focuses on control algorithm development and does not provide automatic generation of IEC 61131-3 compliant PLC nor full automation of project within TIA portal environment. As a consequence, engineers are still required to manually integrate and adapt the developed control logic inside TIA Portal, limiting the overall benefits of automation at the project engineering level.

Another example of industrial process programming [2] propose an approach for automated PLC code generation in the context of building automation systems, where control algorithms based on operating modes are specified using the IFC schema and automatically translated into IEC 61131-3 Structured Text code. This method enables a direct mapping between high-level control models and PLC FBs, improving productivity and consistency at the control logic level. However, this solution still presents limitations linked to manual completion requirements and validation of parameters that are not fully specified at the model level. It focuses on the generation of PLC code but not address the automated creation and maintenance of complete TIA Portal projects, including hardware configuration, I/O mapping, and project-level integration. Some control functions as PID controllers or specialised control blocks still require manual implementation or tuning in the PLC environments, remaining dependent on post-generation validation. Therefore, there is a need for a workflow that automates not only code fragments, but the full TIA project engineering lifecycle.

Thesis objective and contribution

This thesis proposes a specification-driven workflow to generate and update Siemens TIA Portal PLC projects, starting from an industrial process construction in Factory I/O and creation of configuration files Excel/XML. The core contribution is a C# application built on *TIA Portal Openness* that translates user-editable configuration inputs (Excel/XML) into automated engineering actions inside TIA Portal. The tool supports both project creation and update workflows and automates key tasks such

as device and rack configuration, systematic I/O addressing, tag table generation, and the instantiation and reuse of a standardised software architecture (e.g., FBs, data blocks, and a coherent main program structure). The industrial process construction is made in Factory I/O software that is adopted as a digital twin to simulate the physical process controlled by the PLC. This provides a real-time representation of the system enabling the validation and testing of the control software, reducing commissioning time and integration risks.
The objective of this thesis is to provide an automation tool that helps engineers reduce time, effort, and human errors during creation and updates of PLC project in Siemens TIA Portal. To reach this result, the following goals are defined and grouped as follows:

- **Tool design and implementation**
  - Design and implement a modular C# application based on editable configuration files (Excel or XML), exploiting Siemens *TIA Portal Openness* libraries;
  - Enable the automatic creation and update of PLC projects, including hardware configuration (racks and I/O modules), tags, FBs, and instance DBs;
  - Implement import and export functionalities for FBs and instance DBs across different TIA Portal projects.
- **Digital twin and industrial process modelling**
  - Model an industrial process within the Factory I/O environment, including conveyors, load and transfer units, and a Cartesian pick-and-place system;
  - Use the digital twin configuration as input for the automatic generation of the corresponding PLC project.
- **Scalability and evaluation**
  - Demonstrate the scalability of the proposed approach by progressively extending the system from a simple industrial process to a more complex one through the replication and configuration of reusable FBs;
  - Evaluate the effectiveness of the proposed tool in terms of reusability, maintainability, and reduction of manual engineering effort.

The remainder of this thesis is organised as follows: Chapter 2 reviews the state of the art; Chapter 3 presents the proposed methodology and implementation; Chapter 4 reports the experimental validation and discusses results; finally, Chapter 5 concludes the work and outlines future developments.

Chapter 2

State of the art

2.1 Traditional PLC engineering workflows

As introduced in Chapter 1, the traditional PLC programming workflow relies on a manual engineering approach. According to the official Siemens documentation [8], the process typically begins with the creation of a new TIA portal project, where the target PLC is selected by choosing the desired CPU type and firmware version from the hardware catalogue. The automation system is then defined in the device configuration environment by assembling the hardware structure including Rack, CPU and I/O modules, all inserted from the catalogue. Each hardware component is subsequently parametrised to match the physical installation requirements. Communication interfaces are configured by assigning the *PROFINET* device name and network parameters, enabling connectivity with distributed I/O modules. Once the hardware configuration is completed, the project is compiled to ensure consistency between the hardware setup and the software environment. The configuration can then be downloaded to the target device, either a physical PLC or a simulated instance such as PLCsim. After the hardware setup, the software development phase begins. Tag tables are created to define variables and associate them with process addresses, enabling structured access to inputs, outputs, and internal memory areas. The control logic is then implemented through the manual creation of program blocks such as OBs, FBs, FCs and instance DBs. Logic development is carried out using the programming languages defined by the IEC 61131-3 standard, primarily LAD, FBD, and SCL. The choice of language depends on the control requirements and the engineer's preference: graphical languages such as LAD and FBD are commonly used for discrete logic and signal-oriented control, while SCL enables textual and algorithm implementations. All engineering activities are performed within the graphical TIA Portal environment through a point-and-click interaction paradigm. Device insertion, parametrization, block creation, language selection, compilation,

and download require explicit user interaction. The workflow follows a sequential structure in which each phase must be manually completed and verified before proceeding to the next. Compilation feedback must be evaluated by the engineer, and any inconsistencies must be resolved before commissioning. Once the PLC is switched to RUN mode, online commissioning activities are performed, including variable monitoring, signal forcing, and verification of communication with distributed devices. Overall, the traditional engineering process is characterised by a strongly interactive and user-driven workflow. Although this approach provides transparency and fine-grained control over each configuration step, it can become time-consuming and repetitive in projects involving multiple similar devices, recurring architectures, or frequent design iterations. The limitations of this purely interactive and manual engineering paradigm have motivated the investigation of alternative approaches aimed at increasing efficiency, consistency and scalability in PLC software development.

2.2 Automatic PLC code generation approaches

In response to the limitations of manual PLC engineering workflows, together with the increasing complexity of industrial automation systems, the research community and industrial practitioners have increasingly investigated automatic code generation techniques. These approaches aim to reduce manual intervention, minimise human errors and improve consistency, scalability and efficiency, specially in large-scale or repetitive applications. Several methodological directions have emerged in both academia and industry, ranging from model-based engineering to rule-driven configuration frameworks. The following sections review the most relevant categories of automatic PLC code generation approaches, discussing their working principles, advantages and limitations.

2.2.1 Model-Based and Model-Driven Engineering

Model-Based Engineering and Model-Driven Engineering represent one of the most widely investigated paradigms for automatic PLC software generation. In these approaches, the system is first described using high-level models, often expressed in Matlab and *Simulink*, UML, SysML or domain-specific modelling languages. The behavioural and structural aspects of the control system are defined abstractly, independently from the target PLC implementation. Automatic code transformation mechanisms from the model are then used to generate IEC 61131-3 compliant code, such as LAD, FBD, Structured Text. The transformation process is typically based on predefined mapping rules that convert model constructs into executable PLC

blocks. An example of the Model-Based approach [9] propose the implementation of a temperature control system using Matlab and *Simulink* combined with the *Simulink* PLC Coder. The control strategy is first developed and validated at model level within *Simulink* environment. The plan dynamics are analysed using Matlab System identification toolbox and controller parameters are tuned prior to deployment using the PID Tuner toolbox. Once the control performance is validated in simulation, the *Simulink* PLC Coder automatically generates IEC 61131-3 compliant Structured Text code. The generated code is then used to program the HMI in *Codesys* and some modifications are required as the global variables of constant type and adjustments to generate programming code obtained with closed-loop. Finally it is carried out a comparison of the *Simulink* and *Codesys* output curves to determine the percentage of error between them and verify consistency in the simulated behaviour. These resulting percentages of error are shown in the Table 2.1. All the observed errors remain below 3%, confirming the consistency of the implemented model and validating correctness of the resulting simulation.

Table 2.1: Errors between *Simulink* and *Codesys* for heating curve times determined with positive disturbance and setpoint

Extracted from source: [9]

Action of interest	**Error [%]**
Max. initial temperature	0.3010
Max. disruption temperature	2.8652
Max. change setpoint temperature	0.1281
Stabilization	0.0089
Before the disruption	0.3382
Before the setpoint change	0.0602

The primary advantage of this approach is the validation of the control algorithm in a simulation environment prior to deployment, that reduce engineering errors and optimise performance before integration into the PLC. Other advantage is the automatic code generation that ensures consistency between the validated model and the final implementation, reducing manual programming effort and potential human errors. However, this implementation presents some limitations. The generated PLC code is constrained by the capabilities and configuration of the code generation tool, which can limit customization, readability and update of the resulting Structured Text. Another disadvantage is that this kind of approach is particularly well suited for algorithmic or continuous control problems and less applicable to complex discrete event logic or highly modular industrial architectures. The integration with existing PLC design ecosystems may require further adaptation steps, as the code generation process often takes place outside the native design environment.

2.2.2 Template-Based PLC Code Generation

Template-based approaches focus on reusing predefined software architectures and generating PLC code by instantiating configurable templates. Instead of manually creating each program block, engineers define parametrised templates that are automatically filled with data from a plant model or control logic specification to produce IEC 61131-3 compliant PLC programs. This approach can use modular automation concepts and reusable FB libraries, guaranteeing maintainability, speeder code generation and reusability. An example of this kind of PLC Code Generation [1] propose an implementation of model-driven framework that allows modular machine definition and the automatic generation of the corresponding automation project, including the hardware and software configuration. The solution decomposes the machine into standardised functional modules, referred to as stations. Each station is reusable functional unit, associated with predefined interface definitions that includes both hardware and software configurations stored in a structured repository. The machine definition is expressed using configuration languages such as XML, to describe modular machine structures, and AML, used to represent hardware and software configurations. These formats allow hierarchical representation of modular machine with formal description of interfaces and explicit definition of I/O and component relationships. The modular machine defined in XML, together with its hardware configuration and control software, is compiled into a set of reusable mechatronic components. Finally, is generated the project in TIA Portal including PLC hardware configuration, network settings, I/O mapping, control logic instantiation and inter-module interconnections. In order to implement automatic PLC project generation, is used a C# API layer that orchestrate TIA Portal in background. Hardware configurations are exported as AML, while software artefacts are exported to XML through TIA Openness. The generated Hardware and Software artefacts are stored in an XML-native repository and re-imported using TIA Portal import functionalities to compose the final modular machine automation project. During the generation process, stations are instantiated based on the configuration, hardware components are configured and software components are parametrised and interconnected according to the hierarchical structure defined in the configuration model. This overall architecture of the proposed framework is efficiently explained and shown in Figure 2.1.

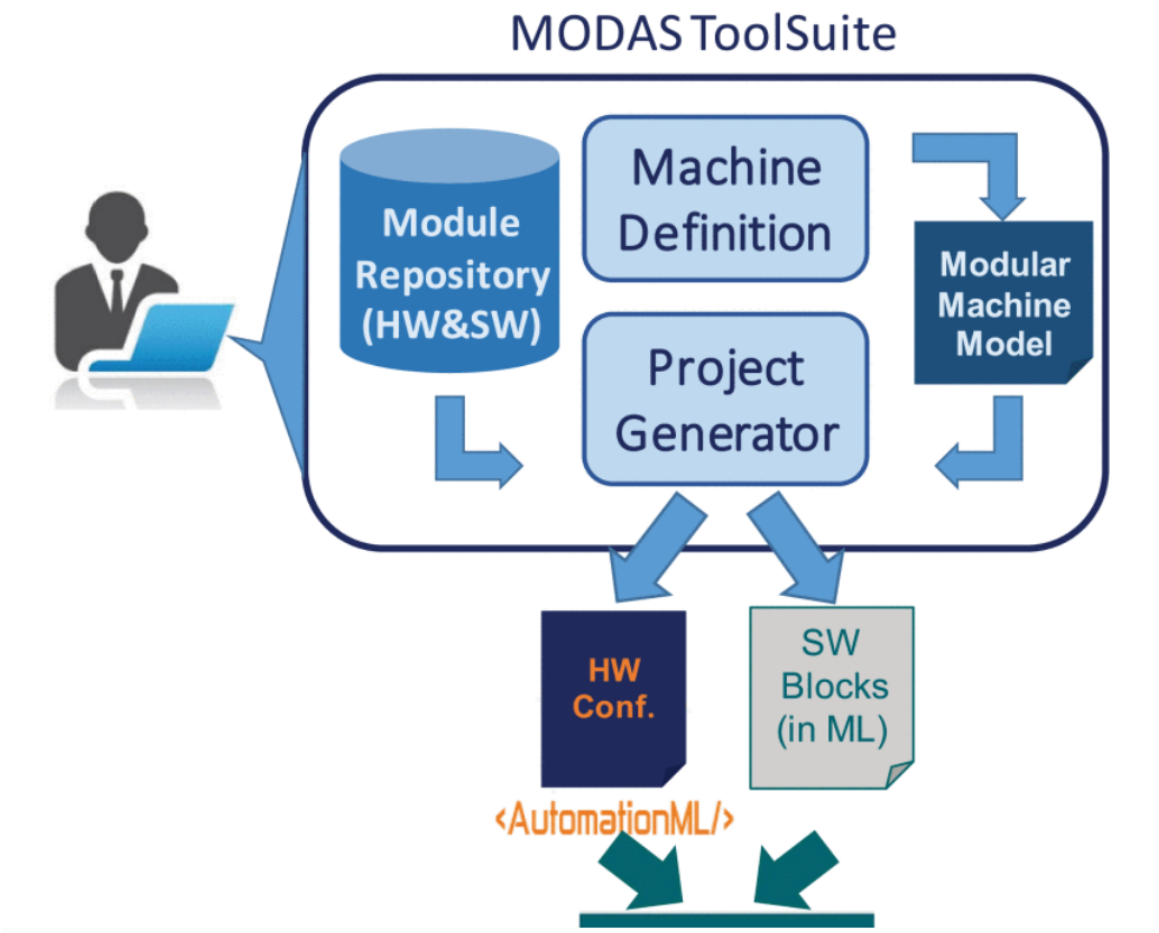


Figure 2.1: Architecture of the MODAS ToolSuite for automatic generation of modular machine automation projects [1].

This approach is validated through a practical case study involving a modular machine configuration, demonstrating the feasibility of automatically generating a consistent and executable PLC project without manual block creation and wiring. The results indicate a substantial reduction in engineering effort and improved architectural consistency across machine variants, thanks to the automatic assembly of predefined PLC modules, standardisation and feasible reuse in modular machines. However, the flexibility of the solution depends heavily on the completeness and maintainability of the module library and XML schema definitions. Is highly effective for standardised modular systems, without the possibility of make modifications in case of evolved systems. This approach supports only the initial creation of PLC project and not the entire lifecycle of a project.

2.2.3 Automatic Control Logic synthesis

Automatic Control Logic Synthesis refers to approaches in which executable PLC programs are automatically derived from a formal behavioural representation of the control system. Most of these methods are used in Supervisory Control Theory and related formal framework [10]. Instead of manually implementing sequential logic, engineers define system behavioural using structured representations, such as finite state machines, Sequential Function Charts, Petri nets or other discrete-event systems. In these approaches, the control behaviour is defined using mathematically rigorous models that describe system states, transitions, synchronization constraints and safety constraints. A synthesis or transformation engine then converts these formal representation into executable PLC code, ensuring behavioural equivalence between the formal model and the generated implementation. Such approaches are particularly relevant in systems characterised by complex sequencing, mode management, or safety-related interlocks, where correctness and consistency of state transitions are critical. Logic synthesis focuses on generating the control behaviour itself, rather than instantiating structural software components. An example of this process [2] proposes a structured pipeline that automatically derives executable PLC code from a mode-based representation of a building energy control system. This approach is based on the concept of mode-based control algorithms, where system behaviour is organised according to discrete operating modes. Each one of these is associated with specific control actions and transition conditions. Rather than implementing these modes directly in IEC 61131-3, the control algorithms are first formalised in a structured representation, referred to as IFC data model. The operational modes, transitions, hierarchical relationships, and control parameters are encoded within an IFC compliant schema, by using Python and *IFCOpenShel* to process the IFC data. This representation serves as a structured behavioural model of the control system, separating the specification of operational logic from its final PLC implementation. Another important aspect is the application of (SIPN) as the formalised description method to ensures the description of the control algorithms without ambiguities and errors. Using JavaScript Object Notation, the IFC data are then translated in PLC program format. Finally, IFC interfaces with the developed software are used to automatically generate PLC code for implementing mode-based control algorithms. The overall workflow of the proposed method, from behavioural specification to automatic PLC code generation, is illustrated in Figure 2.2.

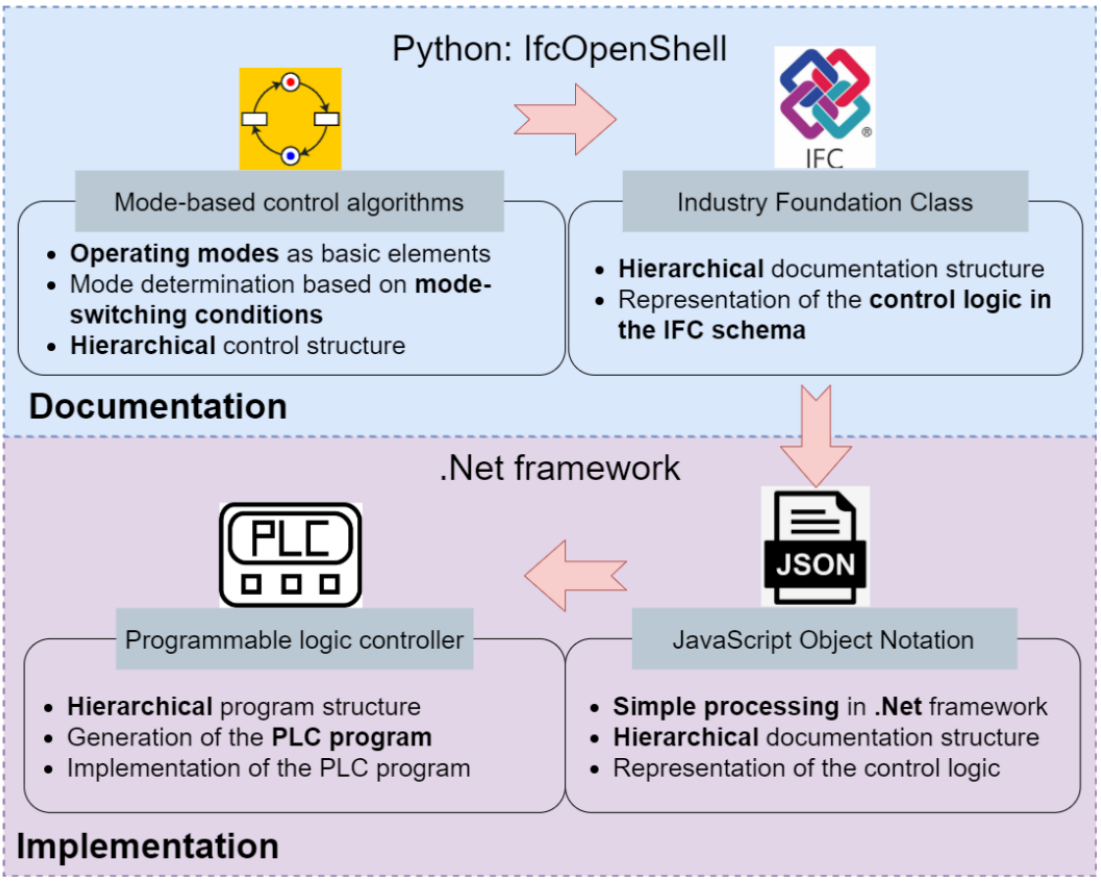


Figure 2.2: Workflow for automatic PLC code generation from mode-based control algorithms using IFC and JSON representations [2].

The method synthesises the behavioural logic itself from a structured representation. The resulting PLC program is deployed and validate in a real building energy system scenario. Several advantages emerge from this approach. By formalizing operating modes within a structured model, the method reduces manual implementation errors in complex state-dependent control systems. The separation between behavioural specification and PLC implementation enhances traceability and maintainability. Furthermore, leveraging IFC enables integration with building information modelling environments, potentially facilitating consistency between system design data and control implementation. However, the approach presents some limitations. The modelling effort required to encode behavioural logic within IFC can be significant, especially for large-scale systems with numerous modes and transitions. The generation process depends on dedicated transformation tools, which may limit portability across different industrial PLC engineering environments. Finally, this approach supports only the initial creation of PLC project and not the entire lifecycle of a project and scalability may become challenging as hierarchical mode structures grow in complexity.

2.2.4 Rule-Based and configuration-Based approaches

Rule-Based and configuration-Based approaches to automatic PLC code generation focus on deriving executable control software from structured configuration data and predefined transformation rules. These methods operate on structured descriptions of system components, process parameters, and connectivity information, which are interpreted by a generation engine according to a predefined rule set. Typically, the system is described through machine configuration artifacts, as structured data models, parameter sets, semantic descriptions, that capture the composition of products units, communication relationships, and operational constraints. Then a rule engine or transformation framework processes this configuration and instantiates corresponding PLC software components, parameter assignments, and interconnections. The logic is derived through deterministic mapping rules embedded in the generation framework. An example of this approach [3] proposes a method that enables orchestration of production operations in an automated production system, allowing a PLC to orchestrate rapidly reconfigurable system, support changes in physical and control configurations or support the introduction of new products. The implementation is delivered through the software application PCG that generates PLC source code from structured production inputs, as The Bill of Equipment and the Bill of Process, by constructing an Abstract Syntax Tree and applying predefined transformation rules. The generated PLC code is then validated using a Siemens hardware and software stack comprised of *Tecnomatix Process Simulate*, TIA Portal, and S71500 PLCs. The overall workflow of the PCG framework is illustrated in Figure 2.3.

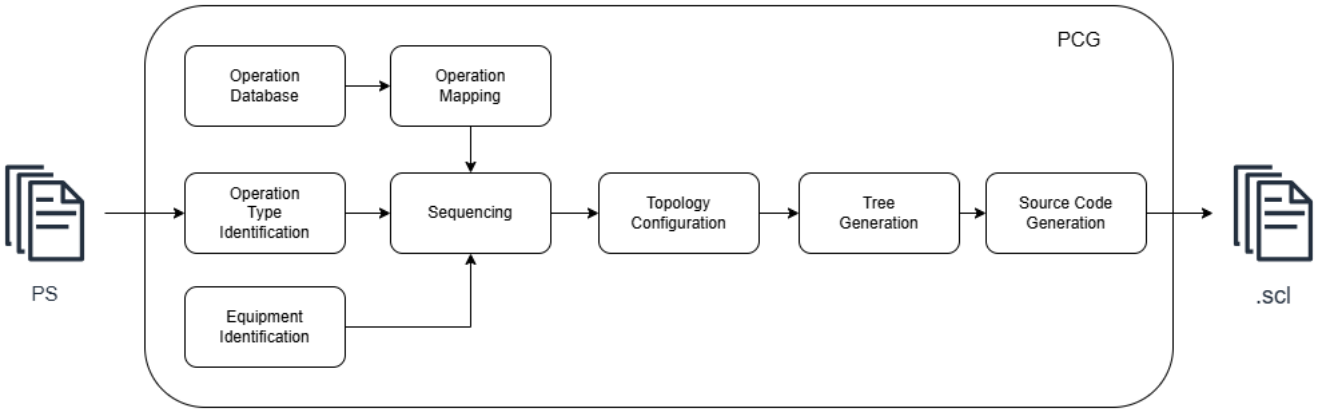


Figure 2.3: Workflow of the PCG for automatic PLC source code generation from production system descriptions [3].

This method enables rapid reconfiguration of manufacturing systems and is validated on a robotic inspection process running on real industrial hardware within the *Omnifactory* demonstrator. This approach significantly reduces development time,

likelihood of human errors, and improves consistency in production environments that are reconfigurable, dynamic and interconnected. Furthermore, is applicable to real industrial PLC projects and by relying on high-level system models and structured production inputs, the method allows the control logic to adapt automatically to changing production configurations. This increases scalability, make easier the management of complex production systems and supports flexible manufacturing scenarios in which both the physical layout and the control structure evolve over time. On the other hand, this approach remains dependent on predefined synthesis rules and well-structured configuration inputs focusing primarily on the code generation phase and does not support the entire engineering lifecycle. Additionally, depends on the availability and correctness of high-level system models and structured configuration inputs. The quality of the generated PLC code is directly linked to the accuracy and completeness of these models. Finally, scalability in very large or highly production systems introduce complexity in rule management and model maintenance. As the system grows, ensuring consistency between models, configuration data, and generated control logic require additional engineering effort.

2.3 Specification languages and engineer-friendly process descriptions

An important use in automatic PLC code generation is referred to specification languages and engineer-friendly process descriptions that can help engineers in creation and update of configuration settings. Specially in case of complex and high-scalable systems with a lot of components, or evolving systems where very often are applied modifications. The aim is to describe system configurations and control logic using structured accessible formats. Examples of this type files are Excel, CSV, XML, DSL, used in order to easily set configuration data, import/export data, update and make modifications reducing human errors, make the system more maintainable and adaptable. In industrial automation, Excel and CSV-based configuration approaches are very simple and familiar to use. They can be used to define I/O mappings, device lists, signal parameters, module and FBs configurations or tags with their addresses. These structured files act as intermediate configuration artifacts that can be parsed by software tools to automatically generate PLC variables, FB instances, or communication mappings. Domain-Specific Languages (DSLs) can be used to provide higher-level textual descriptions of control behaviour or machine configuration. A DSL allows to describe sequences, modules, interlocks, or process relationships using a specified syntax. The DSL specification is then interpreted into executable PLC code. Within industrial research literature, the key characteristic of these methods is that the PLC project is not programmed manu-

ally but derived from this structured description through a generation tool. This is very important in contexts where repetitive architectures, standardised machine variants, or large I/O configurations are involved. However, the flexibility of such approaches is limited by the expressive power of the chosen specification format. In this thesis, and in general in PLC engineering environments such as Siemens TIA Portal, configuration-driven specification approaches can be integrated with programmatic project manipulation frameworks. For instance, in the project of this thesis, structured configuration files are parsed by external software application developed in C#, which uses *TIA Portal Openness* APIs to automatically create or update hardware configurations, FBs, and tag tables within an existing project. The configuration Excel file, in this case, becomes the high-level specification artifact, while the Openness interface acts as the execution mechanism that materialises the described system within the engineering environment. This integration represents a practical evolution of specification-driven automation, combining engineer-friendly configuration artifacts with official API-based project manipulation.

2.4 I/O mapping, tagging, semantic alignment and traceability

According to model-based engineering approaches adopted in industrial context, the automation project should be derived from a coherent semantic model in which devices, signals, and control logic are not treated as isolated elements, but as semantically connected entities within a unified engineering data structure. I/O mapping represents the formal association between physical inputs/outputs and their corresponding logical representations in the PLC program. In this context also tagging plays a central role. When naming conventions, I/O mapping and structured tag tables are defined, tags become part of a semantic layer that supports automation and validation. Semantic alignment is very important in this processes because allows to preserve the meaning across engineering domains such as electrical design, logical part, HMI configuration and simulation environment. Without semantic alignment, inconsistencies between engineering domains can arise, leading to integration errors and costly commissioning phases. Also traceability mechanisms contribute to preventing and resolving inconsistencies across engineering domains that must remain semantically integrated. By maintaining explicit relationships between signals, devices and control artifacts, traceability enables impact analysis, automated consistency checks, structured change management, and cross-domain consistency verification.

2.5 TIA Portal Openness

In the context of automatic PLC code generation, *TIA Portal Openness* provides an API to TIA Portal engineering environment, for automating engineering tasks [11]. It provides objected-oriented API access to TIA Portal with different software solution tools for data exchange, project planning, engineering, verification and online services. Through the *Siemens.Engineering.dll* assembly, Openness provides access to projects, devices, hardware configurations, PLC software, blocks, tag tables and HMI elements. The internal structure of the project presents a hierarchical object model in which Device objects act as container elements for `DeviceItem` instances, reflecting the same logical hierarchy shown in the TIA Portal hardware and network views. An important property for automation process is referred to the programmatic import and export of configuration data via CAx and AML mechanisms. Beyond data exchange, the Openness object model provides structured access to PLC software elements through high-level groups such as PLCBlocks and PLCTypes, which support import and compile functions. This allows an external C# application to generate blocks and data types and trigger compilation processes, verifying the syntactic and structural correctness of automatically generated code. Openness also enables access to hardware-level configuration attributes, including IO controller and IO system interface parameters. Additionally, Siemens allows long-term stability of the API and cross-version compatibility of the *Siemens.Engineering.dll* assemblies, maintaining stability across TIA Portal releases. For this thesis aim, *TIA Portal Openness* is used as the execution part that construct the configuration data into fully instantiated TIA Portal projects. As mentioned before, a C# application processes externally defined configuration files (Excel) and programmatically creates or updates devices, PLC tags, blocks, and related engineering artifacts as instance DBs. The generated project remains consistent with the internal TIA engineering structure, ensuring semantic alignment, I/O coherence and traceability throughout the automated workflow.

2.6 Digital Twins and Factory I/O

In recent years, in industrial automation environments are been adopted digital twin technologies. These are virtual representation of physical systems, capable to replicate their structure and behaviour through simulation models. In the context of PLC engineering, digital twins are very useful to validate implemented control logic, I/O behaviour and sequencing strategies before deploying the system in the physical plant. In this thesis is used Factory I/O as digital twin solution. It provides a realistic simulation environment in which sensors and actuators can be modelled and connected to a PLC in real time, allowing thorough testing of automation strategies

under safe and controlled conditions. As a result, are minimised commissioning time, on-site debugging and the risk of unexpected issues.Although digital twins as Factory I/O, improve validation and testing phases, they do not eliminate inconsistencies or scalability limitations within the PLC project generation workflow. This means that programmable engineering interfaces have to be capable of materializing structured specifications directly into executable PLC projects, ensuring semantic alignment and structural coherence between simulation and implementation. This is a very important step in this thesis about the connection and consistency between simulation and logic control environments.

2.7 Comparative analysis of reviewed literature

Table 2.2: Comparison of the main characteristics of PLC generation methodologies discussed in the literature and the proposed solution

Approach/Ref.	Model-Based/ [9]	Template-Based/ [1]	Logic Synthesis/ [2]	Rule-Based/ [3]	Proposed approach
Main input	Simulink model; analysed plant dynamics; tuned control parameters	XML machine definition, AML HW/SW descriptions; module repository	IFC-based behavioural model; mode hierarchy; SIPN representation; JSON transformation	Bill of Equipment; Bill of Process; structured production descriptions	Excel configuration files; JSON mappings; Factory I/O driver data

Continued on the next page

Approach/Ref.	Model-Based/ [9]	Template-Based/ [1]	Logic Synthesis/ [2]	Rule-Based/ [3]	Proposed approach
Generated artefacts	IEC 61131-3 Structured Text control code	HW configuration; SW modules; I/O mapping; project composition in TIA Portal	PLC behavioural control logic	PLC source code derived from production operations and topology	HW configuration; PLC tag tables; FBs; instance DBs; main logic; project update in TIA Portal
Only code generation	Yes	No	Yes	Yes	No
Project structure	No	Yes	No	No	Yes
Platform dependence	Simulink PLC Coder and external Matlab modelling tools	predefined repositories, TIA Portal Openness for project composition	IFC modelling and dedicated transformation tools	PCG framework and structured production models	Siemens TIA Portal Openness
Update support	No	Limited	No	Limited	Yes

Table 2.2 shows several common characteristics and differences among the PLC generation methodologies discussed in the literature with respect to the aspects most relevant to this thesis. A first common trend across most approaches is the attempt to raise the abstraction level of PLC development by introducing intermediate representations between the engineering specification and the final control implementation. All approaches aim to reduce manual PLC programming by transforming higher-level descriptions into executable control logic. Another com-

mon advantage observed across several methods is the potential improvement in engineering consistency. By deriving PLC programs from structured models or configuration data, these approaches reduce the likelihood of manual programming errors and improve traceability and consistency between system specification and implementation. In particular, this advantage is evident in model-based and logic synthesis methods, where the control logic is derived directly from validated behavioural descriptions. The table highlights important distinction about the type of engineering artefacts produced by the different approaches.The model-based approach presented in [9] focuses mainly on the generation of IEC 61131-3 control code from validated Simulink models. Similarly, the logic synthesis approach discussed in [2] derives behavioural PLC logic from formal mode-based descriptions. The rule-based method proposed in [3] also concentrates primarily on the generation of PLC source code from structured production models. In all these cases, the main output is the control logic itself, rather than the complete PLC project structure. By contrast, the template-based framework proposed in [1] moves beyond code generation and supports the automatic composition of broader project artefacts, including hardware and software structures. However, this approach is mainly oriented toward the initial assembly of modular machine projects based on predefined repositories and configuration schemas. As a result, it provides limited support for the controlled evolution of an already existing PLC engineering project after the first generation step. A second relevant difference concerns the type of input specification and its accessibility for engineering practice. Model-based and logic synthesis approaches rely on specialised modelling environments, such as Simulink or IFC-based formal representations, which are effective for behavioural specification but are less accessible for engineers who need to configure and update automation projects through simpler process-oriented artefacts. Rule-based approaches improve this aspect by relying on structured production descriptions, but they still remain strongly dependent on dedicated transformation frameworks and domain-specific data structures. Template-based approaches provide stronger modularity, but their flexibility depends on the completeness of predefined repositories and interface definitions. Another important comparison criterion is the degree of support for project lifecycle management. Most reviewed approaches automate either the generation of control code or the assembly of predefined project elements, but they provide little or no support for the incremental update of an already generated PLC project. This is particularly relevant in industrial contexts where process configurations evolve over time and engineering changes must be introduced without recreating the project from scratch. Overall, the reviewed literature shows that previous approaches make important contributions in code generation, formal correctness, modular reuse, or production reconfigurability. However, they tend to address these aspects separately. In particular, none of the reviewed contributions

simultaneously combines engineer-friendly configuration artefacts, direct generation of PLC project structures, incremental update of existing projects, and tight integration with the native Siemens TIA Portal engineering environment.

Table 2.3: Summary of the main advantages, disadvantages and limitations of the reviewed literature references

Approach/Ref.	**Advantages**	**Disadvantages**	**Limitations**
Model-Based/ [9]	Early validation through simulation; automatic generation of IEC 61131-3 code; consistency between validated model and generated code	Dependence on external modelling and code generation tools; limited readability and customisation of generated code	Mainly suited for algorithmic control; does not support full PLC project structure generation or project update
Template-Based/ [1]	Modularity and reuse of automation components; automatic generation of HW and SW structures	Dependence on predefined module libraries, repositories and configuration schemas	Limited flexibility when extending the architecture; mainly supports initial project assembly rather than lifecycle updates
Logic Synthesis/ [2]	Formal behavioural specification; improved traceability between system design and control logic	High modelling effort; specialised modelling frameworks required	Limited integration with industrial engineering environments; focused on behavioural logic rather than full project generation

Continued on the next page

Approach/Ref.	Advantages	Disadvantages	Limitations
Rule-Based/ [3]	Rapid reconfiguration of manufacturing systems; scalable generation of PLC code from production descriptions	Strong dependence on configuration data quality and predefined rule definitions	Focused mainly on code generation rather than on complete PLC project lifecycle management

Table 2.3 shows several common characteristics among the reviewed literature references. A common advantage of all the analysed contributions is the reduction of manual PLC programming effort through the use of structured specifications, predefined rules, or formal behavioural representations. Both model-based and synthesis approaches emphasise the benefits of formal system representations for improving traceability between system design and implementation. Similarly, model-based and rule-based methods introduce the advantages of modularity and configuration-based engineering to support scalable automation systems. Despite these benefits, several limitations appear across the analysed works. They all focus mainly on the generation of PLC control logic rather than on the automation of the complete engineering workflow. As a result, tasks such as project maintenance, system updates and integration with existing PLC engineering environments often remain partially manual. Another common limitation concerns dependence on specific modelling tools, configuration frameworks, or predefined libraries. While these mechanisms enable automation, they can also introduce constraints when adapting the approach to new system architectures or evolving production requirements. Therefore, while the literature demonstrates substantial progress in the automation of PLC software development, it still lacks an integrated solution capable of combining accessible configuration inputs, full project-level generation, incremental update mechanisms, and direct manipulation of Siemens engineering projects within TIA Portal.

2.8 Gap analysis

The reviewed literature shows that significant progress has been achieved in the automation of PLC software engineering. Model-based approaches improve the validation of control algorithms before deployment, template-based methods strengthen modular reuse, logic synthesis techniques provide formal correctness for behavioural control, and rule-based approaches support the automatic generation of PLC logic from structured process descriptions. However, several limitations remain open across these methodologies.

- Many approaches are primarily focused on generating PLC logic rather than on constructing and maintaining the complete engineering project structure. As a consequence, important engineering activities such as tag table generation, hardware/software integration, and project-level consistency management often remain only partially addressed.
- The support for project evolution after the initial generation step is generally limited. In many practical industrial scenarios, PLC projects must be updated incrementally to reflect changes in process configuration, additional units, modified I/O mappings, or revised control requirements. The reviewed approaches rarely treat controlled project update as a central capability.
- Configuration accessibility remains an important challenge. Several methods rely on specialised modelling environments, formal representations, or dedicated transformation frameworks that may not be easily usable by automation engineers in day-to-day industrial practice. More accessible specification mechanisms are particularly important in engineering contexts where projects must be adapted, extended, or maintained frequently.
- Direct integration with native PLC engineering environments at project level is only partially covered in the reviewed literature. While some approaches generate code or reusable modules, fewer works address the automated manipulation of complete engineering artefacts within industrial platforms such as Siemens TIA Portal.

These observations reveal a methodological gap between existing research on PLC automation and the practical needs of industrial engineering workflows. In particular, there is still a lack of approaches capable of combining engineer-friendly configuration artefacts, project-level generation, controlled project update, and direct integration with native Siemens engineering environments.

2.9 Positioning of the thesis contribution

In this context, the contribution of this thesis is positioned at the intersection of configuration-driven engineering, project-level PLC generation, and programmable manipulation of Siemens TIA Portal projects. Unlike approaches that focus mainly on the generation of control logic, the proposed framework aims to generate and update a broader set of engineering artefacts, including hardware configuration, PLC tag tables, function blocks, instance data blocks, and main control logic. Unlike approaches centred on initial project assembly only, the framework also introduces

an incremental update workflow that modifies an already existing PLC project according to revised process specifications. The proposed methodology also differs from model-centric and formal-specification approaches by relying on engineer-friendly configuration artefacts, such as Excel and JSON files, which are easier to adapt in practical industrial settings. Finally, through the use of Siemens TIA Portal Openness, the framework directly manipulates engineering artefacts inside the native Siemens environment rather than stopping at external code generation. For these reasons, the novelty of this thesis does not lie only in automating PLC code generation, but in proposing a configuration-driven framework capable of supporting both the creation and controlled evolution of Siemens PLC projects while preserving I/O consistency, structural coherence, and integration with the simulation environment.

Chapter 3

Methodology

3.1 Methodological framework and pipeline

The work proposed in this thesis is based on a structured methodological framework that transforms a virtual process specification into an instantiated and maintainable PLC project. Instead of using a traditional PLC engineering workflow, which is typically manual, time-consuming and error-prone, the proposed approach implements a transformation pipeline developed in C#. In this pipeline, hardware configuration, process data and the automatic generation of FBs are integrated within a coherent workflow. The methodological framework is organized into a sequence of distinct steps. The process begins with the modelling of the industrial system in the Factory I/O virtual environment, where the configuration device is defined and saved. In the next step, configuration Excel files are prepared to define hardware settings, FBs informations and the structure of the main control logic. A dedicated C# application is then implemented by leveraging TIA Portal Openness. This application acquires all configuration data as input and, together with parameters provided by the user, transforms the process specification into a structured project within the TIA Portal environment. User inputs are provided through a command prompt interface that automatically opens when the C# application is executed. The methodology supports the automatic creation of a new project and the update of an existing one. The update flow ensures synchronisation between evolving process specifications and the corresponding PLC project. This enables controlled evolution of the automation system without requiring manual restructuring or duplication of engineering artifacts. Finally, the generated TIA Portal project is validated through simulation of the process in Factory I/O, verifying the correct execution of the industrial process and the proper interaction between sensors, actuators and control logic. The overall methodological framework and the associated transformation pipeline are illustrated in Figure 3.1.

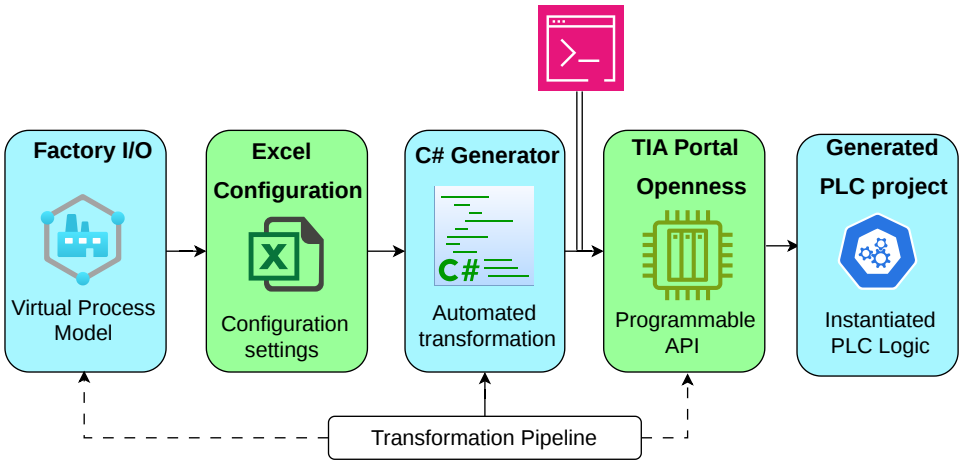


Figure 3.1: Methodological framework and pipeline [4]

3.2 Modelling in Factory I/O

The methodological framework, as shown in Figure 3.1, begins with the modelling of the industrial processes within the Factory I/O simulation environment. As discussed in the previous chapter, Factory I/O is used as a digital twin platform in which sensors and actuators are defined according to the devices added to the environment. Sensors and actuators are linked to specific addresses in order to reproduce the behaviour of a real automation system. The simulation environment represents the structural basis of the PLC project, as it defines the physical and logical I/O configuration that will later be implemented in TIA portal project thorough the automatic generation of the tag table.

3.2.1 Industrial processes design

Two different simulation environments are developed in Factory I/O. The first one, shown in Figure 3.2, is based on a pre-designed model for the distribution of boxes sorted by height [12].

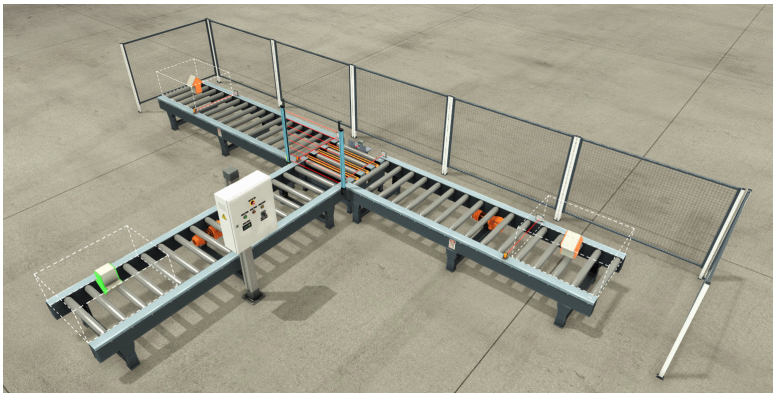


Figure 3.2: Sorting by Height pre-designed model [4].

In the original configuration, boxes of different heights are transported along a conveyor system and sorted according to their dimensions. The list of transported boxes within this system is illustrated in Figure 3.3.

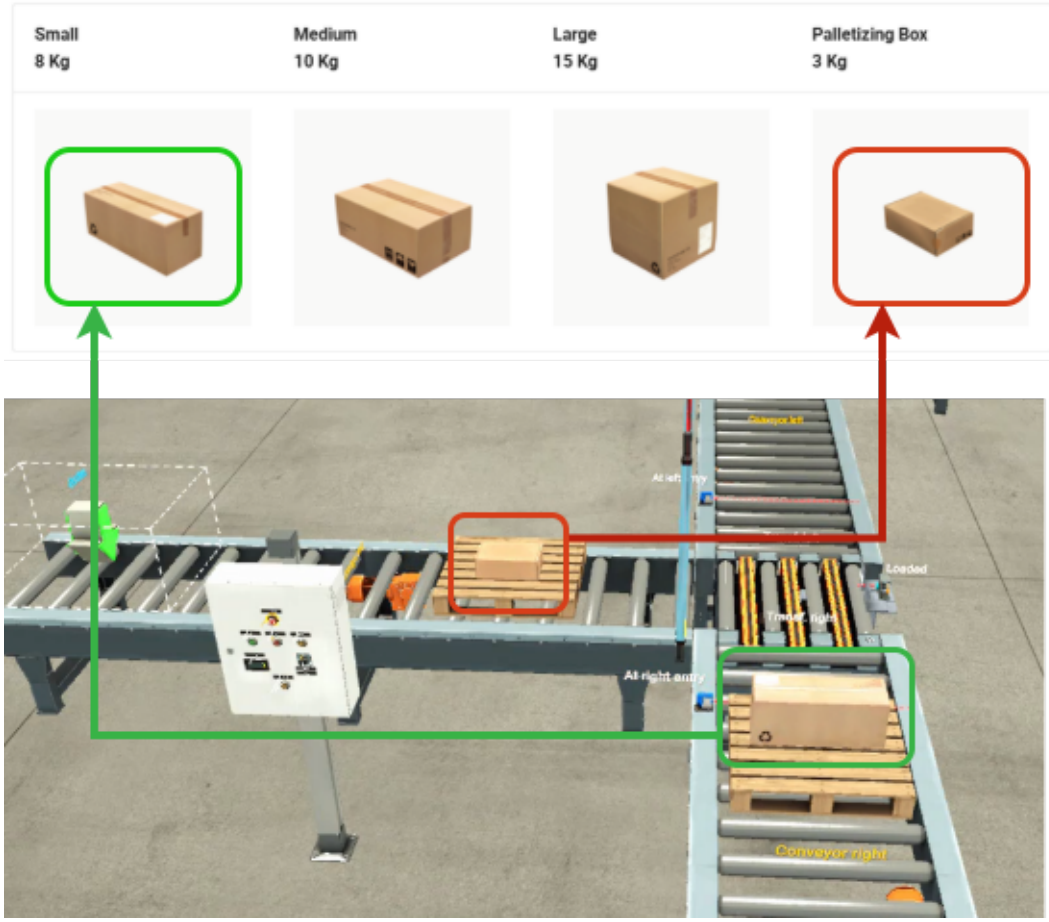


Figure 3.3: Transported boxes [4]

Starting from this preconfigured model, the industrial process is expanded by introducing an additional conveyor on the left-hand side, along with a two-axis Pick&Place system and a manual button, as shown in Figure 3.4.

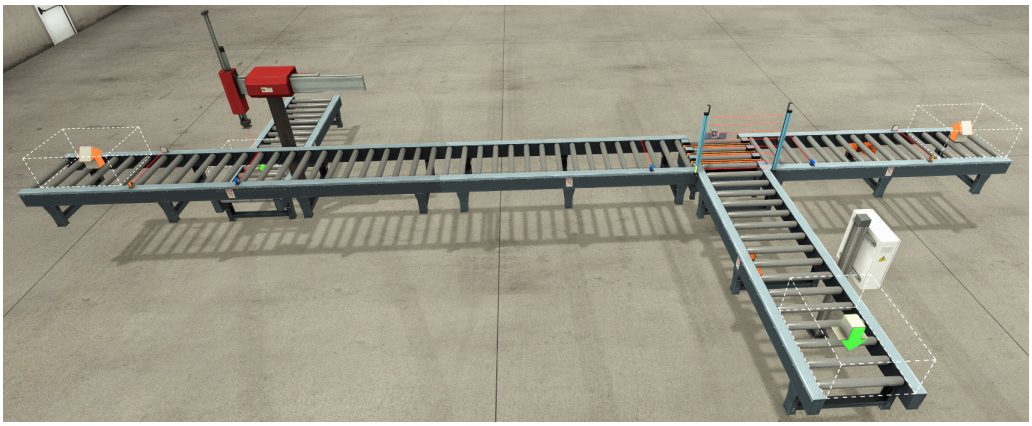
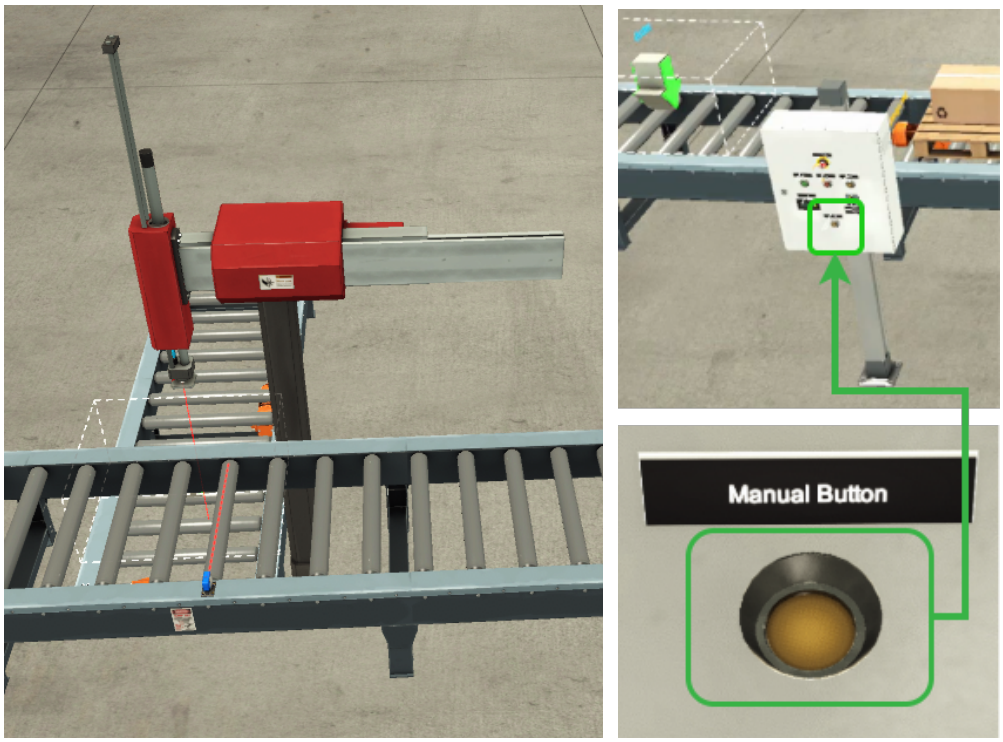


Figure 3.4: First industrial process design [4]

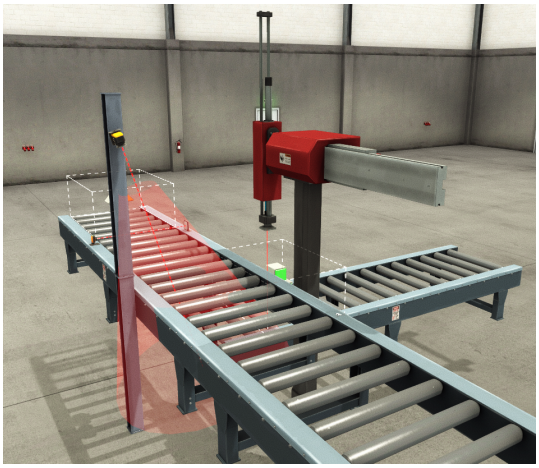
The new units added are shown in Figure 3.5. Specifically, the Pick&Place system and the diffuse sensor are shown in Figure 3.5a, while the manual button used for interaction with the operator is shown in Figure 3.5b.



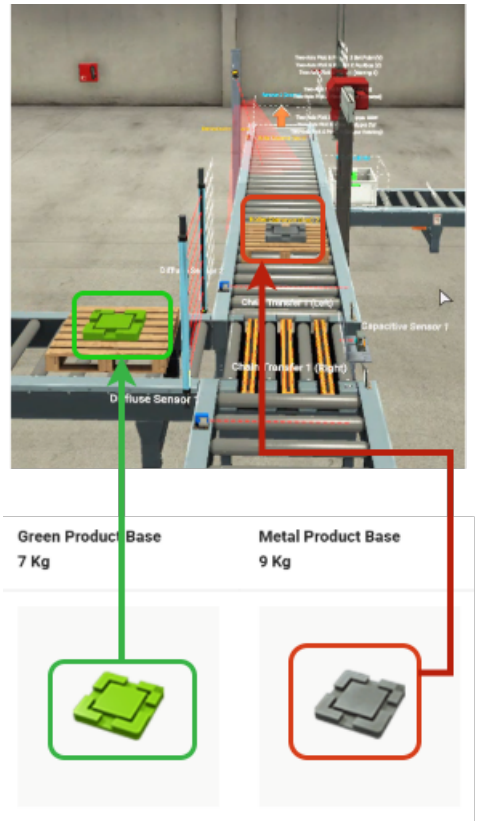
(a) Sensor and Pick&Place [4] (b) Manual Button [4]

Figure 3.5: Inserted units in the first industrial process [4]

The purpose of the manual button is to allow the operator to decide whether a box can continue along the left-hand conveyor. If the operator presses the button, the box proceeds along the left-hand conveyor. Otherwise, the box is deemed unsuitable and the Pick&Place system is activated to remove it from the left-hand conveyor. A second simulation environment is therefore designed to operate completely autonomously. In this case, no manual button is used and the activation of the Pick&Place is entirely managed by the automatic control logic. To implement this autonomous process, the model of the left-hand side is slightly modified. The left-hand conveyor carries both metallic and non-metallic objects, which are distinguished by different colours: grey objects represent metallic items, while green ones correspond to non-metallic ones. These items transported on the left-hand conveyor are shown in Figure 3.6b. The additional components introduced for the second autonomous process are shown in Figure 3.6a.



(a) Vision sensor and Pick&Place [4]



(b) Transported objects on the left conveyor [4]

Figure 3.6: Inserted units in the second industrial process [4]

3.2.2 Driver configuration

Once all the components of the industrial process have been added to the simulation environment, the driver configuration can be created. In this phase, all actuators and sensors associated with the industrial components are configured. After defining the

driver interface by specifying the number of analogue and digital inputs and outputs, the user can perform the I/O mapping by connecting each used sensor and actuator to a specific address of the driver configuration. Once the driver interface is completed, it is exported in XML format and saved in a shared folder between the local PC and the virtual machine. This file is later used as an input for C# application, which automatically constructs the tag table in the generated TIA Portal project. The driver configuration of the first industrial process is shown in Figure 3.7, while the configuration associated with the second process is presented in Figure 3.8.

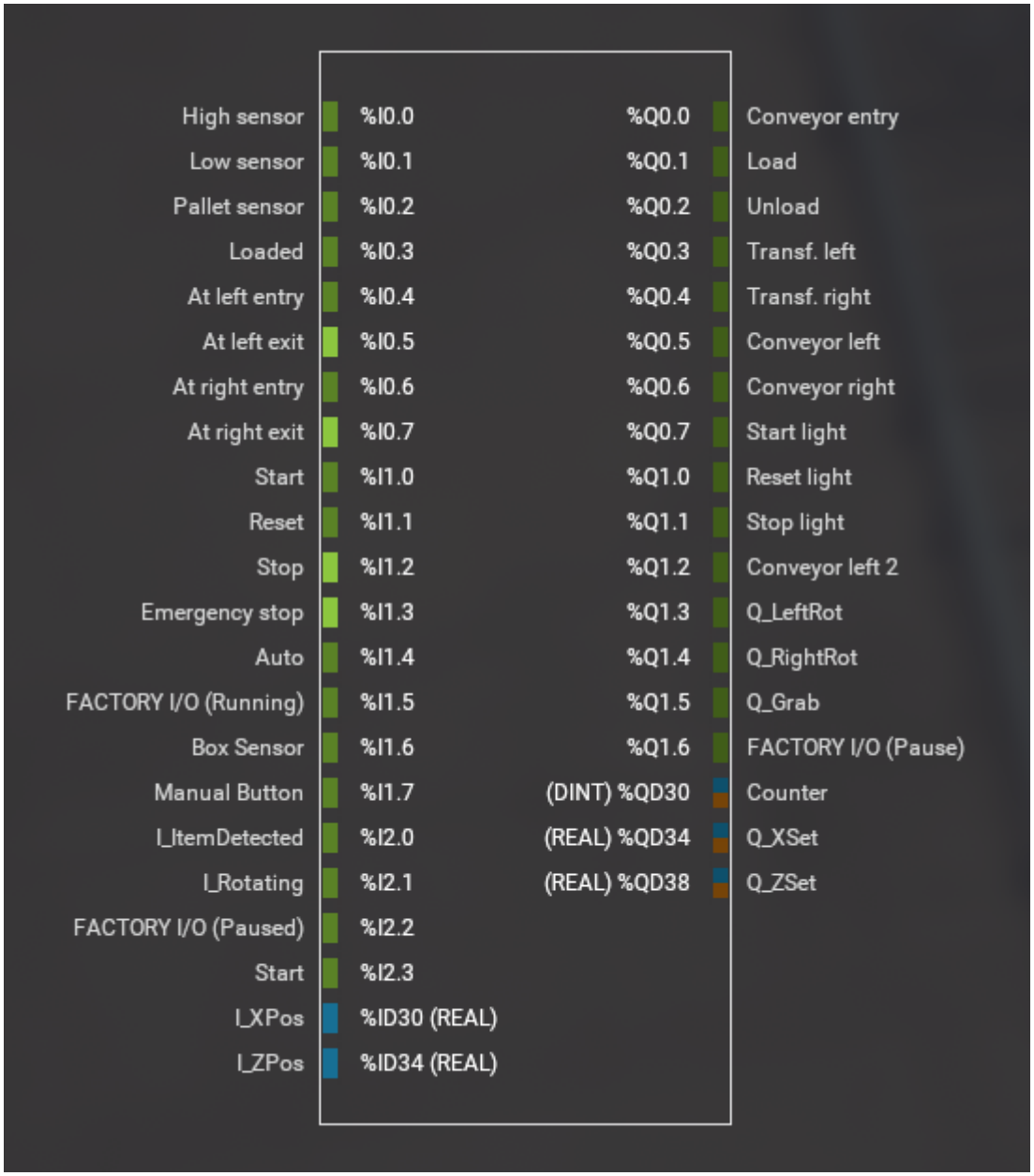


Figure 3.7: Driver configuration of the first industrial process in Factory I/O [4] [4]

Figure 3.8: Driver configuration of the second industrial process in Factory I/O [4] [4]

3.3 Configuration and system setup

The next step of the methodological framework concerns the definition of structured configuration inputs and the preparation of the execution environment. This phase acts as the bridge between the simulation model developed in Factory I/O and the automatic generation of the PLC project. In this step, the system specification is formalised through configuration artifacts that describe both hardware settings and software logic structures. These artifacts are then processed by the C# generator, which interprets the configuration data and produces the corresponding elements within the TIA Portal project.

3.3.1 Excel-based configuration files

The configuration part of the framework relies on structured Excel files used as high-level specification artifacts. Each industrial process is associated with a configuration file composed of multiple sheets describing hardware configuration (CPU and I/O modules), tag definitions and the state-machine logic of the FBs (Conveyor, Pick&Place, LoadTransfer and Main logic units). The overall structure of the configuration file is summarised in Table 3.1. The first two sheets define the hardware configuration, while the remaining sheets describe the software components of the

Table 3.1: Structure of the Excel configuration file

Hardware		**Software**							
Sheet1	**Sheet2**	**Sheet3**	**Sheet4**	**Sheet5**	**Sheet6**	**Sheet7**	**Sheet8**	**Sheet9**	**Sheet10**
CPU	I/O modules	Tags	Test FB	Pick & Place	Conveyor Distribution	Load Transfer	Conveyor Infeed	Conveyor Recycling	Main

automation system.
Starting from the first industrial process modelled in the Factory I/O simulation environment, the hardware part is configured by specifying the CPU device in the first sheet and additional I/O modules (if required) in the second sheet. In the industrial process considered in this thesis, no additional I/O modules are necessary because the selected CPU provides all required inputs and outputs. This configuration is just consistent with the addressing scheme defined in both Factory I/O and TIA Portal. The first sheet configuration for this industrial process is reported in Table 3.2.

Table 3.2: First Sheet: CPU configuration in the Excel file

deviceName	**deviceItemName**	**typeIdentifier**
PLC_1	CPU 1211C DC/DC/DC A	OrderNumber:6ES7 211-1AE40-0XB0/V4.1

When additional modules are required, they can be specified in the second sheet, as illustrated in Table 3.3.

Table 3.3: Example of second Sheet: additional I/O module configuration

deviceName	**deviceItemName**	**typeIdentifier**
Module AI	Mod_AI_1	OrderNumber:6ES7 531-7KF00-0AB0/V2.0
Module AI	Mod_AI_2	OrderNumber:6ES7 531-7KF00-0AB0/V2.0
Module DI	Mod_DI/DO	OrderNumber:6ES7 521-1BH10-0AA0/V1.0

Tag definitions can be specified in the third sheet of the Excel file. However, in the framework developed in this thesis, the tag table is automatically generated by parsing the XML configuration file exported from Factory I/O. This ensures consistency between the simulation environment and the generated TIA Portal project. However, the Excel tag sheet remains available as an alternative or additional configuration method for automatic tag table generation in TIA portal. Table 3.4 shows an example of tag specification within the Excel configuration file.

Table 3.4: Example of third Sheet: tag configuration

Name	**DataType**	**LogicaLADdress**	**Comment**
start	Bool	%I0.0	start button
mem	Bool	%M0.0	memory bit
stop	Bool	%Q0.0	stop button
temp	Real	%QD34	temperature

The remaining sheets define the state-machine logic of the FBs that implement the behaviour of the industrial units, including Conveyor, Pick&Place, LoadTransfer and the main control logic. An example of the configuration used for the LoadTransfer FB is shown in Table 3.5.

Table 3.5: Seventh sheet: state-machine configuration for the LoadTransfer FB

step	**Out Condition**	**actions**	**TransCondition**	**next Step**
0	I_pallet	Q_StartLoad:=TRUE	I_LowBox AND I_HighBox	1
			I_LowBox AND NOT I_HighBox	2
1	I_Loaded	Q_TransfRight:=TRUE	I_AtRightExit	3
2	I_Loaded	Q_TransfLefT:=TRUE	I_AtLeftExit	3
3		Q_StartLoad:=FALSE; Q_TransfRight:=FALSE; Q_TransfLefT:=FALSE	I_AtLeftExit OR I_AtRightExit	0

Before finalising the structure of the configurations for the finite-state machines, two alternative representations in Excel were evaluated for the automatic generation of PLC code. The two configurations differed in the way the information was organised within the various columns and rows of the Excel spreadsheet. A series of tests was carried out to assess the execution time of the code generation process for each configuration. Based on the results of these tests, the configuration structure that ensures the most efficient generation process was selected. The same configuration format was then adopted for the implementation of all FBs used in the automation system. As shown in Table 3.5, the configuration describes the device's state machine using five columns. The *step* and *nextStep* columns represent the current state and the next state of the system. State transitions are governed by the *TransCondition* column, which defines the logical conditions that trigger the transition. The *OutCondition* column specifies the input conditions associated with the current state, whilst the *actions* column defines the outputs activated during the execution of that state. This configuration structure is used consistently for all finite-

state machines implemented in the automation system. For replicated units, such as conveyors, different configuration schemes are defined depending on the specific logical flow required for each type of conveyor, including the input conveyor, the distribution conveyor and the recycling conveyor.

A complete description of the Excel configuration conventions is provided in Appendix A.

3.3.2 Additional settings and execution parameters

In addition to the Excel configuration file, the C# generator requires additional runtime parameters defined in a JSON configuration file. This file is very easy to manage and update, and contains the general settings required for the framework to function correctly. The configuration parameters include:

- TIA Portal version.
- Path of the Excel configuration file.
- Sheet index mapping for configuration parsing.
- Path of the reference project downloaded from the Factory. I/O official documentation.
- Name of the communication FC block connecting Factory I/O and PLCsim.
- Path used to store the exported XML file for Factory I/O communication
- Number and name of Factory FC block to be imported in the current project.
- Default project folder path.

A complete structure of the JSON configuration file used by the generator is reported in Appendix B.

3.3.3 Development and deployment environment

The framework operates across two different environments.

- **Local development environment** The C# application is developed using Visual Studio Code and compiled into an executable file.
- **Virtual machine environment** The virtual machine hosts the industrial automation software stack, including TIA Portal, PLCsim and Factory I/O. Through a shared folder between the local machine and the virtual machine, the compiled executable file can be executed inside the virtual environment.

The shared folder enables the exchange of configuration files, Factory I/O exported data and the deployment of the compiled executable used to generate the TIA Portal project.

3.4 C# application architecture

The following step of the methodological framework concerns the implementation of the C# executable application responsible for the automated generation and update of PLC projects in TIA Portal. The application is implemented as a modular console-based software system that orchestrates configuration management, user interaction and programmable engineering operations through TIA Portal Openness. The code structure is layered and modular making the C# application easy to manage and modify by users that want to adapt it to their usage case. This approach improves code maintainability and readability, facilitates debugging and testing activities, and allows the framework to be extended with additional functionalities. The modular structure promotes also the reuse of individual services, guaranteeing the scalability of the proposed approach and long-term maintainability. Figure 3.9 illustrates the overall modular structure of the C# code.

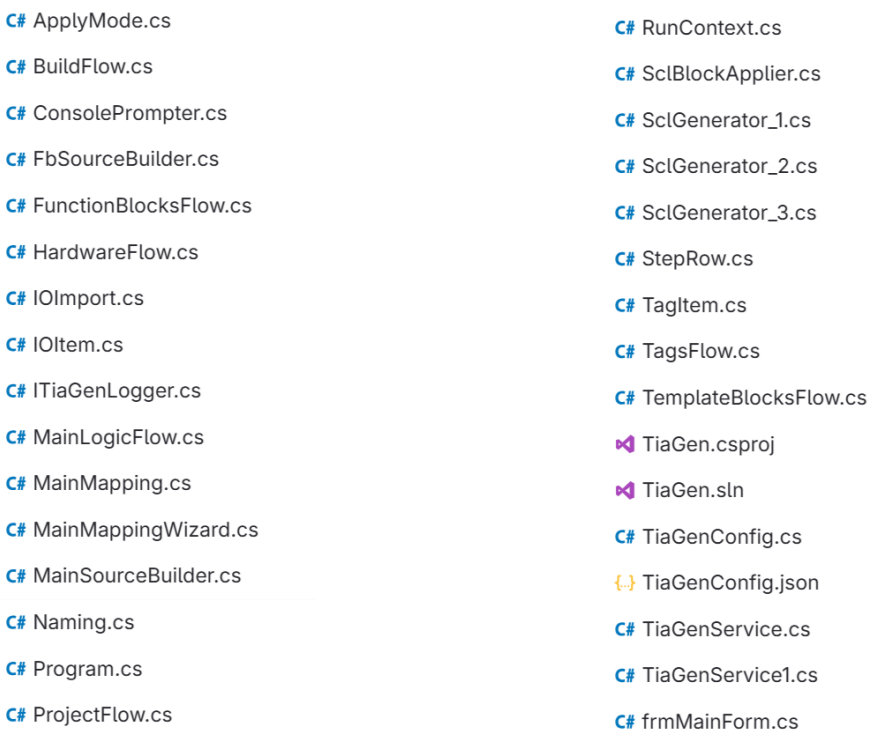


Figure 3.9: C# modular code structure [4]

3.4.1 Initialisation and configuration management

The execution of the generator starts with an initialisation phase in which all configuration parameters, software dependencies and execution services are prepared before entering the main execution stage. Configuration parameters are stored in a structured JSON file that contains all the information required by the generator, including the TIA Portal version, the paths of the configuration files and the indexes of the Excel sheets used for the process specification. The JSON configuration file is parsed and loaded into the dedicated configuration class `TIAgenConf.cs`, which provides a centralised access point for all configuration parameters used by the application. The initialisation logic is implemented in the `Program.cs` file. During this stage, the JSON configuration file is loaded and the parameters are mapped into the corresponding fields of the configuration class. In particular, the generator retrieves the version of TIA Portal to be used during the execution through the parameter `config.TiaVersion`. This mechanism allows the framework to adapt the execution environment to the installed version of TIA Portal. In the current implementation, the default configuration uses TIA Portal version V19, but is possible to configure more versions and then specify what to use in the configuration file. After loading the configuration parameters, the application initialises the required dependencies associated with TIA Portal Openness, including the assembly resolver and the necessary engineering libraries. Subsequently, the logging system and the generator service are instantiated, enabling the application to start the execution phase of the PLC project generation. The generator service coordinates several specialised modules that implement the different steps of the automated project generation process, including hardware configuration, tag generation, template block management, function block generation and main logic construction.

3.4.2 Interactive user-driven parameters

Once the generator service has been created, the application launches TIA Portal in the background and opens a command-line interface to interact with the user. The generator follows a "human-in-the-loop" approach, in which the user supervises and controls key engineering decisions during the automation process. After the initialisation phase, the command line prompts the user to select the generator's operating mode. Specifically, the user can choose whether to create a new TIA Portal project or update an existing one. This choice is managed by the `ApplyMode.cs` class, which sets the execution mode to `Create` or `Upsert`. The selected mode determines how the generator handles subsequent engineering operations, such as creating new elements or updating existing project components. During this interaction phase, the system collects additional execution parameters required for the

automated generation process. This approach allows the framework to remain flexible, while maintaining consistency with the predefined configuration rules. Furthermore, it allows for the selective regeneration of project components without requiring a complete rebuild of the PLC project. The initialisation of the generator service and the user's first interaction in the PLC process are illustrated in Figure 3.10.

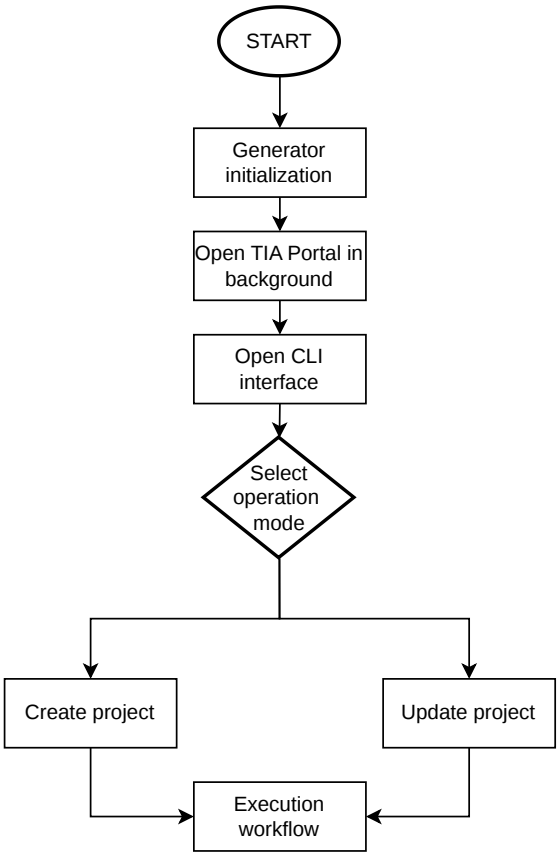


Figure 3.10: Generator service initialisation and first user interaction [4]

3.4.3 Command line interface functionalities

The command-line interface provides a wizard that guides the user through the main design stages required to generate or update the PLC project. Using this interface, the user can monitor the generator's operation and, if desired, enable specific design operations. The command prompt guides the user through a structured sequence of decisions:

1. Creation of a new project or opening of an existing one.
2. CPU configuration, allowing the user to insert a new CPU device or detect an existing one within the project.

3. Optional insertion of additional I/O modules into the hardware rack.

4. Import of tags generated from the Factory I/O driver configuration. In update mode, the existing tag table can be removed and recreated to ensure consistency between the simulation environment and the PLC project.

5. Export and import of communication blocks used for the connection between Factory I/O and PLCsim. The user can also export arbitrary FC, FB, or instance DB blocks from other TIA Portal projects and import them into the current project.

6. Automatic generation of FBs and their associated instance instance DBs implementing the control logic of the industrial units, including pick-and-place systems, load-transfer mechanisms and different conveyor types.

Once the design phases are complete, the generator automatically compiles and saves the project. The main logic block responsible for orchestrating the execution of the generated functional blocks is then created, and a final compilation phase is performed to ensure the project's consistency. The complete sequence of design operations performed by the generator is implemented within the `Run()` method of the `TiaGenService` class. The execution workflow implemented in the `Run()` method of the generator service is shown in Figure 3.11.

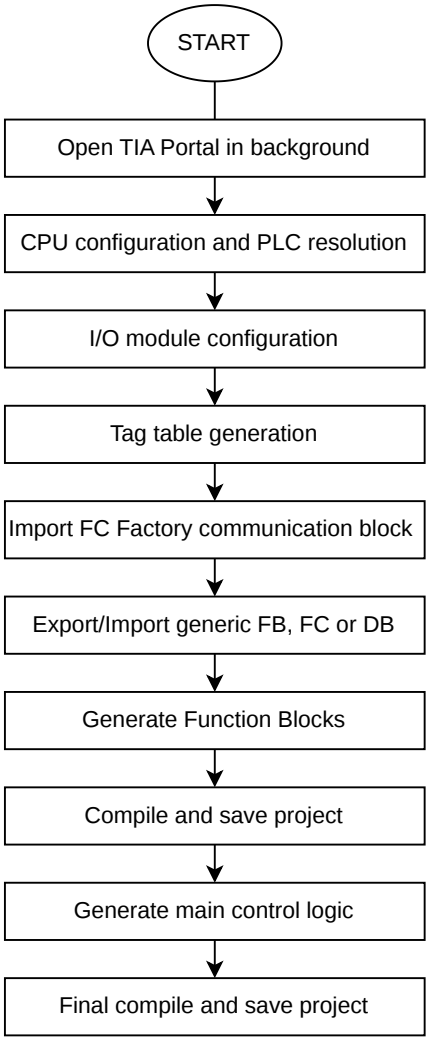


Figure 3.11: Execution workflow implemented in the `Run()` method of the generator service [4]

3.5 Transformation logic and mapping rules

A central aspect of the proposed methodology is the deterministic transformation of structured configuration artefacts into PLC engineering elements within the TIA Portal project. In order to improve reproducibility, the transformation process can be described as a set of explicit mapping rules linking the simulation environment, the configuration files, and the generated software artefacts. The framework operates on three main input sources: the Factory I/O driver configuration exported in XML format, the Excel specification file describing hardware and control logic, and the JSON configuration file containing execution settings and path definitions. These inputs are parsed by the C# generator and translated into hardware and soft-

ware artefacts according to predefined rules. From a methodological perspective, the transformation process is organised into the following stages:

1. Parse the JSON configuration file and load execution settings, including paths, TIA Portal version, and sheet mappings.
2. Analyse the Factory I/O XML driver configuration and extract the list of sensors, actuators, signal names, data types, and logical addresses.
3. Parse the Excel sheets associated with hardware configuration and function block logic.
4. Generate or update the hardware configuration in TIA Portal according to the hardware specification.
5. Generate the PLC tag table from the XML driver configuration or, alternatively, from the Excel tag sheet.
6. Generate the FBs and their corresponding instance DBs from the state-machine specifications defined in Excel.
7. Generate the main logic and map FB interface parameters to PLC tags, constants, or expressions through the interactive mapping wizard.
8. Compile and save the project in order to validate the generated artefacts.

This rule-based transformation process ensures that the generated PLC project remains structurally aligned with both the virtual process specification and the external configuration artefacts.

3.5.1 Input parsing and artefact generation rules

The transformation between input artefacts and generated PLC elements follows explicit mapping rules. The Factory I/O XML driver configuration is used as the primary source for PLC tag generation. Each exported signal is parsed and mapped to a corresponding PLC tag according to the following rule:

Factory I/O signal $\rightarrow$ *PLC tag name, data type, logical address*

The detailed generation workflow implemented by the class `TagsFlow.cs` is described in Appendix C.
The Excel hardware sheets define the PLC device and, if required, additional I/O modules. Each row in the hardware sheets is interpreted as a hardware instantiation rule:

deviceName, deviceItemName, typeIdentifier → TIA Portal device or device item creation

The detailed generation workflow implemented by the class `HardwareFlow.cs` is described in Appendix C.
The Excel software sheets define the control logic of each industrial unit through a state-machine representation. For each configured unit, the corresponding sheet is parsed and transformed into a dedicated function block and its associated instance data block according to the following rule:

State-machine sheet → FB interface + SCL code + instance DB

The detailed generation workflow implemented by the classes `FunctionBlocksFlow.cs` `FBSourceBuilder.cs` and `SclGenerator_3.cs` is described in Appendix C.
The Main Logic sheet is interpreted as a call-level orchestration specification. Each row identifies a function block instance to be called and the associated interface parameters to be mapped. The final variable assignment is completed through the interactive wizard and persisted into a JSON mapping file, which enables traceable reuse of the selected mappings during subsequent executions.

3.5.2 Naming and interface generation rules

To preserve consistency and reproducibility across generated projects, the framework adopts deterministic naming rules for software artefacts. For each industrial unit, a dedicated function block is generated using the unit type as the base identifier. When multiple units of the same type are instantiated, numerical suffixes are added in order to preserve uniqueness. For example, replicated conveyor units are generated as `Conveyor`, `Conveyor2`, `Conveyor3`, and so on. The associated instance data blocks follow the same convention, using the FB name combined with the suffix `_DB` or an equivalent indexed identifier. The interface of each generated function block is derived from the Excel state-machine specification. Input and output variables are generated from the signal names referenced in the conditions and actions columns. This guarantees consistency between the SCL body of the block and the declared interface variables. The use of deterministic naming conventions provides two main benefits. First, it ensures that generated artefacts can be identified and reused consistently during project updates. Second, it reduces the likelihood of naming conflicts when multiple instances of the same logical unit are created within the same project.

3.5.3 Main logic mapping strategy

The generation of the main control logic follows a semi-automatic mapping strategy. The overall call structure of the main program is derived automatically from the Excel specification, while the assignment of actual variables to FB interface parameters is supervised by the user through the interactive command-line wizard. For each detected FB instance and for each input, output, or in-out parameter, the wizard supports the following mapping options:

- assignment to an existing PLC tag,
- assignment to a constant value,
- assignment to a logical expression,
- no assignment, leaving the parameter temporarily unbound.

The selected mappings are then serialised into a dedicated JSON file. This file acts as a persistent trace of the user-defined associations and enables reuse during subsequent executions, especially in update mode. As a result, the framework combines deterministic automatic generation with controlled user supervision in the final I/O binding phase.
The detailed procedure implemented by the classes `MainLogicFlow.cs`, `MainMappingWizard.cs`, and `MainSourceBuilder.cs` is described in Appendix D.

3.5.4 Consistency preservation and conflict handling

The methodology also includes rules for preserving project consistency and for handling common conflict situations during generation and update. In the case of tag generation, the framework checks whether a logical address is already occupied before creating a new PLC tag. Occupied addresses are skipped in order to avoid duplication conflicts and to preserve consistency with already existing project elements. In the case of function block generation, the update mode distinguishes between the creation of a new block and the regeneration of an existing one. When the user selects an already existing block for update, the framework regenerates the FB and the associated instance DB in a coordinated way, ensuring alignment between the block interface and the internal data structure. If expected configuration inputs are missing, such as an unavailable Excel sheet, an invalid XML path, or an inconsistent JSON parameter, the corresponding generation step cannot be completed correctly. For this reason, the methodology assumes that configuration artefacts are validated before execution. Compilation and saving are used as final consistency checks to detect unresolved structural or syntactic issues in the generated project. A further consistency issue arises during project update, when the

existing main logic still refers to previous versions of the FB interfaces. For this reason, an intermediate compilation step is performed before regenerating the main logic, and a final recompilation is executed after the mapping has been updated. This two-step compilation strategy is necessary to restore consistency between the updated blocks and the orchestration logic of the project.

Algorithm 1 Simplified generation and update procedure

1. Load JSON execution settings
2. Load Excel configuration sheets
3. Load Factory I/O XML driver configuration
4. Open TIA Portal session
5. If mode = Create:
   (a) Create new project
   (b) Configure hardware
   (c) Generate tag table
   (d) Import required communication blocks
   (e) Generate FBs and instance DBs
   (f) Compile and save
   (g) Generate main logic
   (h) Compile and save
6. If mode = Update:
   (a) Open existing project
   (b) Regenerate tag table
   (c) Update or create FBs and instance DBs
   (d) Compile and save
   (e) Regenerate main logic
   (f) Compile and save

The complete update and consistency management strategy implemented by the framework is described in Appendix E.

3.6 Automated PLC project generation via TIA Portal Openness

The C# application interacts with the TIA Portal OpennessAPI in order to programmatically create or update PLC projects. Through this interface, engineering operations that are normally performed manually within the TIA Portal graphical environment can be executed automatically. Based on the user selection performed through the CLI, the application activates one of the two execution pipelines. The first pipeline corresponds to the project creation flow, which generates a new PLC project in TIA Portal from the provided configuration files. The second pipeline corresponds to the project update flow, which modifies an already generated project by introducing new logic elements or updating existing ones. In the context of this thesis, the project creation flow is used to generate the first automation project corresponding to the initial industrial process. Subsequently, the update flow is used to modify the previously generated project in order to implement the second industrial process. The software components involved in the generator service for both pipelines, creation and update, are illustrated in Table 3.6.

Table 3.6: Software components involved in the generator service for both pipelines

Generation step	**Main class**
Project initialisation	`ProjectFlow.cs`
Hardware configuration	`HardwareFlow.cs`
Tag table generation	`TagsFlow.cs`
Block import/export	`TemplateBlocksFlow.cs`
Function block generation	`FunctionBlocksFlow.cs`
Main logic creation	`MainLogicFlow.cs`
Compilation and saving	`BuildFlow.cs`

3.6.1 Project creation flow

The project creation flow represents the first execution pipeline activated by the C# generator service. This pipeline is triggered when the user chooses the creation of a new PLC project via the interactive CLI. The objective of this process is to automatically instantiate a structured TIA Portal project starting from the configuration settings defined in the Excel and JSON files. Through a sequence of automated steps, the system initialises the engineering environment, configures the hardware, generates the required software components and builds the main control logic.

3.6.1.1 TIA Portal instance initialisation

The first step of the project creation pipeline consists in the initialisation of the TIA Portal engineering environment. The generator application instantiates a TIA Portal session through the OpennessAPI. The software is opened in background mode, allowing the generator to interact with the project without requiring direct graphical interaction from the user. Once the environment is initialised and user has selected the creation mode, a new project is created and automatically named using the pattern: *MyNewProject_(date)_(time)*. This naming strategy ensures that each generated project has a unique identifier and avoids conflicts with previously existing projects stored in the same default directory. Before executing the generator, it is very important to verify that the configuration files are correctly defined. In particular, the JSON configuration file specifies the paths of the required resources, while the Excel configuration file defines the hardware and software parameters associated with the industrial process. These configuration files establish the engineering environment in which the automated project generation is performed.

3.6.1.2 Hardware configuration

Once the project has been created, the generator proceeds with the configuration of the PLC hardware. Through the CLI, the user can decide whether to insert a CPU device and additional I/O modules into the project hardware configuration. In particular, the user has the possibility to configure the rack by specifying the desired CPU model in the Excel file by providing the corresponding Siemens order number and version. In this implementation, the selected PLC device is the Siemens `S7-1211C DC/DC/DC` CPU, identified by the order number: `6ES7 211-1AE40-0XB0/V4.1`. In this project, no additional I/O expansion modules are required for the PLC rack configuration. This design choice is motivated by two technical considerations. First, the industrial process implemented in Factory I/O does not require additional extended I/O modules, the digital and analogue I/O count of the selected CPU type is sufficient to handle all defined sensors and actuators in the considered scenario. Second, maintaining hardware consistency with the Factory I/O sample configuration ensures stable communication with Siemens PLCsim. The selected CPU model corresponds to the hardware configuration used in the reference Factory I/O example project, where only the function enabling the connection between Factory I/O and PLCsim is configured.

3.6.1.3 Tag table generation

After completing the hardware configuration stage, the generator proceeds with the creation of the PLC tag table. The user can choose to manage the tag table from the

XML file. At the beginning of project implementation, the tag table was managed from Excel configuration as other settings. In this approach, each tag required the explicit specification of its name, data type, logical address and descriptive comment, so that each digital or analogue I/O was assigned to specific address in the logic memory. At a later time, another solution was chosen that proved to be more consistent, scalable, time-efficient and less prone to errors. The new solution, as previously mentioned, consists in exporting the driver configuration from Factory I/O in a XML file, specifying the path of this file in the JSON configuration. This file contains the mapping between sensors, actuators and the corresponding PLC memory addresses. During execution, the generator parses the XML configuration and automatically creates the corresponding tag table inside the TIA Portal project through the OpennessAPI. This solution ensures consistency between the simulation environment and the software configuration. It is also a scalable approach because, before creating the tag table, all addresses occupied by any existing tags are checked and skipped. For all other free addresses associated with sensors and actuators in the driver configuration, tags are created accordingly.

3.6.1.4 Export and Import of FBs and DBs

The generator also supports the reuse of existing PLC logic modules through an export and import mechanism from other TIA Portal projects. Using the CLI, the user can choose to export the FC from the sample Factory I/O project, after downloading it in the default path. By specifying the name, type and path in the JSON file, the application can export the FC from the other project to an XML file, whose path is specified in the JSON. The function is then imported in the current project that the user is creating. This ensures that the new project will be able to realise connection between Factory I/O and PLCsim. The export and import functionalities are implemented also for arbitrary FC, FB or instance DB. This means that user can decide if export a function or database from any other TIA Portal project, only specifying the corresponding path. In this situation the application opens the other project in background, shows a list of all existing blocks and allows the user to choose what to export in XML file. After it has been exported, the user can decide also to import it in the current project that is creating. This is a very powerful mechanism to support in case of update functions or reuse other projects function that contains the same inner logic. All these involved operations are shown in the Table 3.7.

Table 3.7: Supported block reuse operations

Operation	Description
Export FB/FC/DB	Export block from external project
Export and Import Factory FC	Export FC from external project and automatically import it in the current generated project
Import FB/FC/DB	Import block into current generated project
XML serialisation	Intermediate exchange format between operations

3.6.1.5 FBs creation

Within the project creation pipeline, the generation of FB represents the phase in which the control logic of each physical unit is instantiated inside TIA Portal using the finite state machine logic defined in the corresponding sheet of the Excel file. Each physical component present in the Factory I/O environment (Conveyor, Pick&Place, LoadTransfer), is modelled in TIA Portal as a dedicated FB and its associated instance DB. This design ensures a modular and scalable structure, where multiple units, with identical internal logic, can be replicated by using the same FB and dedicated instance DB. Through the CLI, the user is repeatedly asked for each unit if wants to manage the FB from Excel file. The question is repeated until the user answers negatively. A crucial aspect of this phase is the Excel writing convention. Each different control logic unit is associated to a specific sheet of the Excel file. The sheet specifications are all configured in JSON code. Each excel sheet is not a generic configuration sheet but a structured specification that follows strict column naming and formatting rules, as explained in the Appendix A. These conventions allow deterministic parsing and automatic transformation into SCL code and interface definitions. From a software architecture perspective, two dedicated C# classes operate on the same Excel source. The first class is responsible for generating the SCL body of the FB. It translates the condition actions logic and the step-transition logic defined in Excel into structured SCL code and injects it directly into the FB. The second class generates the structure of FB interface, declaring input and output parameters which are used to create the instance DB with related variables. This class relies on the same Excel conventions to ensure full consistency between logic and interface. During development, two alternative SCL generators were experimentally implemented and evaluated. The first SCL generator code was based on an initial Excel structure, while the second relied on a refined layout. Performance comparisons showed that the second generator achieved lower solution times while preserving functional requirements. For this reason, it was selected as the final implementation for SCL generator code in all units.

In the case of conveyor, three different logic variants are available, corresponding to three possible internal control strategies. The actual industrial process requires four conveyors in total, one for entry boxes, another for control part on the left side boxes and two additional conveyors for item distribution. The user can select the conveyor type among the three available variants. The entry conveyor is modelled to start when the sensor of pallet is disabled. After first conveyor creation, when the next one is created, the corresponding FB and instance DB are numbered with next free index. This allows to create multiple units of the same type, which is Conveyor in this case. For this specific industrial process are then created the two distribution boxes conveyors, one for right-transferring and another for left-transferring. Each conveyor is modelled to start when the transfer variable is triggered and when is activated the sensor that indicates the completion of LoadTransfer unit in the specific direction. The transfer variable becomes true for the left side in case of small box or becomes true for the right side when the box is high. In this way only one of the two transferring is enabled and the number of transferring boxes on each conveyor is taken into account by using a counter. The internal logic for both of them is the same, so the same FB is replicated two times with two different indexes. In this way the internal logic is the same and uses the identical Excel sheet for configuration, but the two conveyors are separated and then associated to different input and output variables. Finally, is created the last one conveyor with a control logic that enables its motion when left conveyor is moving and the sensor is not detecting a pallet. When this sensor becomes true the conveyor goes in the next state and a timer of three seconds is activated. If the user press the manual button before the timer ends, the conveyor starts again and when the box has overcome the sensor, the conveyor comes back to its initial state. If the button is not pressed, a start signal is sent to the Pick&Place and the conveyor restarts only after having received "done" signal from Pick&Place. For all other units, the interaction with CLI is similar but without logic-type selection. This industrial process requires only one instance for each remaining unit:

- LoadTransfer unit

  When a pallet is detected by the input sensor, the LoadTransfer unit starts by activating the load output. At this stage the system waits until the pallet has been loaded and the corresponding sensor confirms that the operation has been completed. The control logic determines the direction of the transfer according to the type of the box detected by sensors. If the detected box is high, the system activates the right transfer, otherwise it enables the left one. The activation is made by setting the corresponding output in order to enable only one transfer direction at a time. This ensures that the pallet is correctly routed according to its height. The system then waits until the pallet reaches

the exit sensor corresponding to the selected direction. Finally, when pallet leaves the unit, the outputs are reset returning the LoadTransfer unit to its initial state.

- Pick&Place unit

  When the sensor on the second left conveyor detects a new pallet, that transports a small box, and the manual button is not pressed in three seconds, this unit starts. The Pick&Place enabling, as explained before, derives from an output that comes from the last explained conveyor. When the Pick&Place is activated, the rotation towards the left side is activated in order to position the manipulator correctly with respect to the box. Once the correct rotation is reached, the positioning phase begins by moving the end effector towards the pick position. This is achieved by setting the two target coordinates that control the manipulator movement in the workspace. When the box is detected within the working area and the required position is reached, the gripping action is executed by activating the grab. Finally, the rotation in the opposite direction is enabled, the object is released in a stackable box and all outputs are reset. The Pick&Place unit returns an output to indicate the completion of the operation and waits until the sensor no longer detects the pallet, ensuring that the manipulation cycle is fully completed before returning to the initial state. The generated industrial units in TIA Portal environment are shown in the Table 3.8.

Table 3.8: Generated industrial units

Unit	**Function Block**	**Instance DB**
Entry Conveyor	Conveyor	DB_Conveyor
Distribution Conveyor (Right)	Conveyor2	Conveyor2_DB
Distribution Conveyor (Left)	Conveyor3	Conveyor3_DB
Recycling Conveyor	Conveyor4	Conveyor4_DB
LoadTransfer	LoadTransfer	LoadTransfer_DB
Pick&Place	PickAndPlace	PickAndPlace_DB

3.6.1.6 Compilation, saving and Main Logic creation

After the generation of all FBs and their corresponding instance DBs, the pipeline proceeds with compilation and saving phase that is performed twice: once before creating the main logic, and once after its generation. The first save and compile step is required to ensure that TIA Portal fully registers the new created blocks inside the project. In particular, the main program must reference FB instances

through explicit calls and must connect their interface parameters to I/O tags. If the project is not compiled beforehand, the automation layer not detect the presence of the generated FBs and the subsequent main creation step cannot build the call structure and the corresponding I/O mapping. Once the project has been compiled and saved the tool starts the Main Logic generation process. This phase is driven by a dedicated Excel sheet, where each row describes a call statement to a specific FB instance and eventually output conditions. The main creation is implemented by using an interactive mapping wizard executed via CLI. For each FB instance detected from the Main Excel and for each variable (input, output, in-out variable), the user is guided through a prompt-based selection process. The wizard supports different assignment strategies, such as binding a parameter to an I/O tag, assigning constants, defining expressions, or leaving the parameter unbound. This interaction enables semi-automatic configuration: the overall structure is generated automatically, while final mapping decisions remain under the user's control. To guarantee traceability and enable incremental updates, the mapping results are persisted into a dedicated JSON file. This file stores the chosen associations between main program variables and the FB interface parameters, allowing the system to reuse an existing mapping configuration in subsequent executions and to assign only missing connections when new units or parameters are introduced. From an implementation perspective, dedicated software components manage the different stages of this process. These components parse the Excel specification, detect the FB instances that must be called in the main program, execute the interactive mapping wizard, and serialise the resulting configuration into the JSON mapping file. After the main logic has been created, a second save and compile step is executed. This final compilation ensures that the entire project is syntactically correct, consistent, and buildable, concluding the project creation flow.

3.6.2 Project update flow

The project update flow represents the second execution pipeline of the framework. It is activated by the user through the interactive CLI. Unlike the creation flow, which instantiates a new project from scratch, the update pipeline operates on an existing TIA Portal project previously generated by the system. The objective is to modify selected components of the project in response to changes in configuration files (Excel and JSON) corresponding to a new or extended industrial process. The update flow ensures controlled evolution of the PLC project while preserving structural consistency and minimizing manual rework.

3.6.2.1 Configuration settings update

The first stage of the update pipeline consists in modifying the configuration settings in order to adapt the generator to the second industrial process.

- JSON configuration file

  The JSON file must be updated whenever configuration parameters differ between the first and the second process. In particular, this may include the path of the XML driver configuration file exported from Factory I/O, as well as the path of the Excel file used to describe the process configuration. Additional parameters may also be updated if required, such as the Excel sheets associated with the FBs of the different units, the configured TIA Portal version, or other resource paths used by the generator. In the case considered in this work, only the path of the new Factory I/O driver configuration file needs to be modified. All the remaining parameters remain unchanged, allowing the generator to update the previously created project in order to obtain the new process configuration.

- Excel-based configuration

  The Excel configuration file must also be updated to reflect the differences between the two industrial processes. In this phase, the hardware configuration remains unchanged, since the update pipeline operates on an already existing project and preserves its physical configuration. The modifications affect only the internal logic of specific FBs corresponding to the units whose behaviour changes in the second process.

  - Recycling conveyor unit

    In the second industrial process, the manual control button is not used. Consequently, the recycling conveyor logic is modified with respect to the configuration used in the first process. The timer previously used in the control logic is no longer required. Instead, the conveyor starts when the first conveyor on the left side is activated and stops whenever a new pallet is detected by the vision sensor introduced in the second Factory I/O simulation environment.

  - Pick&Place unit

    The internal logic of the Pick&Place FB requires only minor adjustments. In particular, the gripping positions of the manipulator are updated. Since the objects manipulated in the second industrial process differ from those used in the first process, the target coordinates along the X and Z axes must be slightly modified. The overall behaviour of the

unit remains unchanged. The Pick&Place operation is still triggered by a start signal coming from the recycling conveyor. Although the sensor type changes, the logic remains equivalent: the digital value returned by the sensor determines whether the object is metallic or not and therefore whether the Pick&Place operation should be activated. The logic responsible for enabling the Pick&Place operation remains implemented within the recycling conveyor control logic.

3.6.2.2 Tag table upsert

In update mode, when the user chooses to manage the tag table through the CLI, the only available update strategy is an *upsert* operation. Since a new industrial process is created in Factory I/O, a new driver configuration file must be exported from the simulation environment. The path of this file is specified in the JSON configuration file and used as input by the generator. To guarantee consistency between the simulation environment and the PLC configuration, the existing tag table is cleared and recreated using the information contained in the new XML driver configuration file. In this way, all tags associated with sensors and actuators are regenerated according to the updated simulation model.

3.6.2.3 FBs upsert

After the tag table has been regenerated, the FBs must be updated for the units whose logic has been modified in the Excel configuration. In update mode, the application allows the user either to create a new FB with the corresponding instance DB for a new unit or to update an existing block. The supported unit types include the different conveyor variants, the LoadTransfer unit and the Pick&Place unit. When the user selects the update option, the application displays the list of existing blocks available in the project. After selecting the block to be updated, the corresponding FB is regenerated according to the updated Excel specification. The associated instance DB is also updated in order to maintain consistency between the block interface and its internal data structure.

3.6.2.4 Compilation, saving and Main Logic creation

Once the FBs have been updated, the project is compiled and saved in order to obtain the updated blocks with their corresponding I/O interfaces. During this first compilation step, compilation errors are expected. In fact, the existing main logic still contains function calls whose I/O interfaces correspond to the previous version of the blocks and are therefore no longer aligned with the updated process configuration. After this initial compilation and saving phase, the user can choose between

two possible strategies for regenerating the Main Logic:

- Creating the Main Logic from scratch through the interactive mapping wizard, linking the appropriate I/O tags to the updated FB interface variables.
- Reusing the existing Main Logic configuration by updating the previously generated JSON mapping file. In this case, the user can modify the stored I/O associations directly in the local file and then choose to use the existing configuration during the Main Logic generation phase.

Both approaches allow the generator to rebuild the main control logic consistently with the updated industrial process.

Chapter 4

Results

4.1 Results and evaluation of PLC projects generated in TIA Portal

This chapter evaluates the results obtained by applying the proposed automated PLC engineering framework to the considered industrial processes. First, the generated PLC projects obtained in TIA Portal are analysed after the execution of the project creation and project update pipelines. The focus is placed on the resulting project structure, on the generated software elements and on the consistency between the configured logic and the final engineering artefacts. The evaluation focuses on the following aspects:

- Verification that the framework is able to automatically generate a complete PLC project structure from external process specifications.
- Assessment of the capability of the framework to support incremental updates of an existing project after modifications in the industrial process.
- Evaluation of the potential reduction in manual engineering effort compared to traditional PLC engineering workflows.
- Verification that the generated PLC project preserves structural and naming consistency between the simulation environment and the PLC implementation.
- Analysis of the manageability of the generated project structure when the complexity of the process increases.

.

Finally, the generated projects are validated through simulation in Factory I/O, showing the correct interaction between the virtual process, the generated PLC logic

and the communication established through Siemens PLCsim. The execution of the generator produces a structured TIA Portal project containing the hardware configuration, the automatically generated tag table, the imported communication block required for Factory I/O interaction, the generated FBs with their associated instance instance DBs, and the main control logic. The resulting project is organised according to a modular structure in which each industrial unit is represented by a dedicated function block and a corresponding instance data block. This confirms that the proposed framework is able to transform the external process specification into a coherent and reusable PLC software architecture.

4.1.1 Automated generation of the PLC project

After executing the project creation pipeline, the framework automatically generates a complete PLC project containing the engineering artefacts required to implement the first industrial process. In particular, the generated project includes:

- Configured CPU device.
- Imported communication function required for Factory I/O and PLCsim interaction.
- Automatically generated PLC tag table.
- Generated FBs and the associated instance DBs for the industrial units.
- Generated main logic block.

Figure 4.1 shows the overall structure of the generated project in TIA Portal. The project tree clearly highlights the imported communication block, the generated function blocks and the corresponding instance databases.

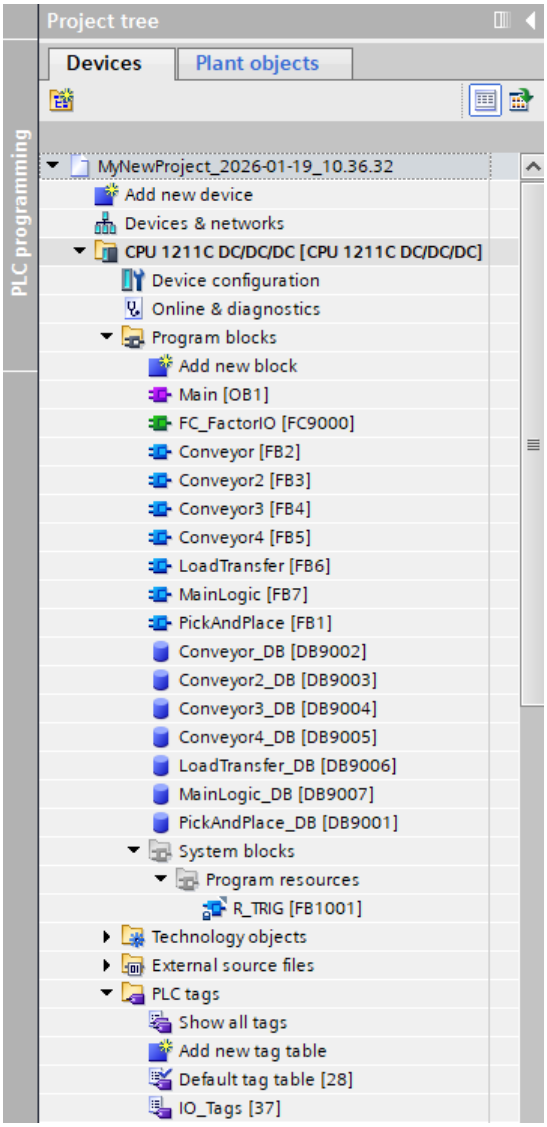


Figure 4.1: Overview of the generated PLC project in TIA Portal after the project creation pipeline [4]

Figure 4.2 confirms that the framework correctly generates the hardware configuration required for the industrial process. As described in the previous chapter, the industrial process requires only the selected CPU, which is instantiated in the rack configuration. The device is automatically created in the first available slot and provides all the integrated digital and analogue I/O resources required for the considered industrial process.

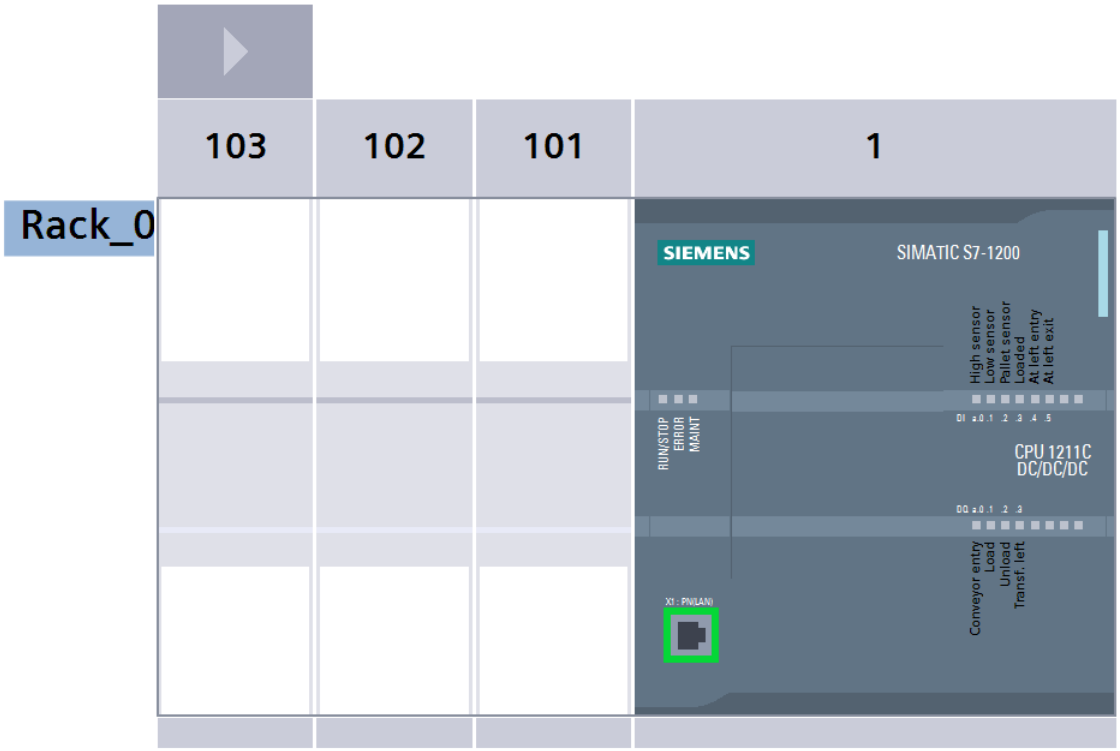


Figure 4.2: Generated hardware configuration [4]

The automatically generated PLC tag table is shown in Figure 4.3. The tags are derived from the Factory I/O driver configuration exported in XML format and include the signal names, data types and logical addresses associated with the sensors and actuators of the simulated process. This result demonstrates the ability of the framework to maintain consistency between the simulation environment and the PLC software configuration. By automatically importing the signal definitions from the simulation driver configuration, the framework reduces the risk of manual addressing errors and naming inconsistencies that typically occur during manual PLC engineering.

IO_Tags

	Name	Data type	Address	Retain	Acces...	Writa...	Visibl...	Comment
1	High sensor	Bool	%I0.0	☐	☑	☑	☑	High sensor
2	Low sensor	Bool	%I0.1	☐	☑	☑	☑	Low sensor
3	Pallet sensor	Bool	%I0.2	☐	☑	☑	☑	Pallet sensor
4	Loaded	Bool	%I0.3	☐	☑	☑	☑	Loaded
5	At left entry	Bool	%I0.4	☐	☑	☑	☑	At left entry
6	At left exit	Bool	%I0.5	☐	☑	☑	☑	At left exit
7	At right entry	Bool	%I0.6	☐	☑	☑	☑	At right entry
8	At right exit	Bool	%I0.7	☐	☑	☑	☑	At right exit
9	Start	Bool	%I1.0	☐	☑	☑	☑	Start
10	Reset	Bool	%I1.1	☐	☑	☑	☑	Reset
11	Stop	Bool	%I1.2	☐	☑	☑	☑	Stop
12	Emergency stop	Bool	%I1.3	☐	☑	☑	☑	Emergency stop
13	Auto	Bool	%I1.4	☐	☑	☑	☑	Auto
14	FACTORY I/O (Running)	Bool	%I1.5	☐	☑	☑	☑	FACTORY I/O (Running)
15	Box Sensor	Bool	%I1.6	☐	☑	☑	☑	Box Sensor
16	Manual Button	Bool	%I1.7	☐	☑	☑	☑	Manual Button
17	I_ItemDetected	Bool	%I2.0	☐	☑	☑	☑	I_ItemDetected
18	I_Rotating	Bool	%I2.1	☐	☑	☑	☑	I_Rotating
19	Conveyor entry	Bool	%Q0.0	☐	☑	☑	☑	Conveyor entry
20	Load	Bool	%Q0.1	☐	☑	☑	☑	Load
21	Unload	Bool	%Q0.2	☐	☑	☑	☑	Unload
22	Transf. left	Bool	%Q0.3	☐	☑	☑	☑	Transf. left
23	Transf. right	Bool	%Q0.4	☐	☑	☑	☑	Transf. right
24	Conveyor left	Bool	%Q0.5	☐	☑	☑	☑	Conveyor left
25	Conveyor right	Bool	%Q0.6	☐	☑	☑	☑	Conveyor right
26	Start light	Bool	%Q0.7	☐	☑	☑	☑	Start light
27	Reset light	Bool	%Q1.0	☐	☑	☑	☑	Reset light
28	Stop light	Bool	%Q1.1	☐	☑	☑	☑	Stop light
29	Conveyor left 2	Bool	%Q1.2	☐	☑	☑	☑	Conveyor left 2
30	Q_LeftRot	Bool	%Q1.3	☐	☑	☑	☑	Q_LeftRot
31	Q_RightRot	Bool	%Q1.4	☐	☑	☑	☑	Q_RightRot
32	Q_Grab	Bool	%Q1.5	☐	☑	☑	☑	Q_Grab
33	I_XPos	Real	%ID30	☐	☑	☑	☑	I_XPos
34	I_ZPos	Real	%ID34	☐	☑	☑	☑	I_ZPos
35	Counter	DInt	%QD30	☐	☑	☑	☑	Counter
36	Q_XSet	Real	%QD34	☐	☑	☑	☑	Q_XSet
37	Q_ZSet	Real	%QD38	☐	☑	☑	☑	Q_ZSet

Figure 4.3: Generated PLC tag table derived from the Factory I/O driver configuration [4]

The result of the automatic function block generation is shown in Figure 4.5 and the corresponding instance DB generation is shown in Figure 4.4. The figures highlight both the generated block interface and the SCL code implementing the state machine logic derived from the Excel specification. This demonstrates that the framework automatically provides the structural artefacts of the project and the internal behavioural logic of the industrial units.

LoadTransfer_DB

	Name	Data type	Start value	Retain	Accessible ...	Writa...	Visible in ...	Setpoint
1	Input			☐	☐	☐	☐	☐
2	I_AtLeftExit	Bool	false	☐	☑	☑	☑	☐
3	I_AtRightExit	Bool	false	☐	☑	☑	☑	☐
4	I_HighBox	Bool	false	☐	☑	☑	☑	☐
5	I_Loaded	Bool	false	☐	☑	☑	☑	☐
6	I_LowBox	Bool	false	☐	☑	☑	☑	☐
7	I_pallet	Bool	false	☐	☑	☑	☑	☐
8	Output			☐	☐	☐	☐	☐
9	Q_StartLoad	Bool	false	☐	☑	☑	☑	☐
10	Q_TransfLefT	Bool	false	☐	☑	☑	☑	☐
11	Q_TransfRight	Bool	false	☐	☑	☑	☑	☐
12	InOut			☐	☐	☐	☐	☐
13	Static			☐	☐	☐	☐	☐
14	Step	Int	0	☐	☑	☑	☑	☐

Figure 4.4: Instance DB generated for the `LoadTransfer` FB [4]

Block interface

```
// *** Code from Excel ***

CASE #Step OF
    0:
        // OUT
        IF #I_pallet THEN
            #Q_StartLoad := TRUE;
        END_IF;

        // TRANS
        IF #I_LowBox AND #I_HighBox THEN
            #Step := 1;
        ELSIF #I_LowBox AND NOT #I_HighBox THEN
            #Step := 2;
        END_IF;

    1:
        // OUT
        IF #I_Loaded THEN
            #Q_TransfRight := TRUE;
        END_IF;

        // TRANS
        IF #I_AtRightExit THEN
            #Step := 3;
        END_IF;
```

```
    2:
        // OUT
        IF #I_Loaded THEN
            #Q_TransfLefT := TRUE;
        END_IF;

        // TRANS
        IF #I_AtLeftExit THEN
            #Step := 3;
        END_IF;

    3:
        // OUT (no condition)
        #Q_StartLoad := FALSE;
        #Q_TransfRight := FALSE;
        #Q_TransfLefT := FALSE;

        // TRANS
        IF #I_AtLeftExit OR #I_AtRightExit THEN
            #Step := 0;
        END_IF;

END_CASE;
```

Figure 4.5: Generated `LoadTransfer` FB showing the SCL implementation of the state machine logic [4]

Finally, the generated main logic is shown in Figure 4.6. The project creation pipeline automatically produces the main control block together with the mapping structure required to orchestrate the generated function blocks. From an engineering perspective, the generated project is almost fully completed after the execution of the automated pipeline. The only remaining manual step consists in inserting the communication FC and the generated main FB into the two networks of the main program OB1, as illustrated in Figure 4.7. This operation is minimal and requires only the placement of already generated blocks from the project tree into the corresponding ladder networks. The decision not to automatically modify OB1 in the current implementation was taken in order to preserve flexibility in the definition of the main control logic. In more complex automation systems, the main control logic may include additional states, alarms, emergency conditions or supervisory behaviours. Managing the main logic through a dedicated function block therefore allows the architecture to remain extensible. Nevertheless, the manual insertion in OB1 represents a current limitation of the proposed workflow and could be further automated in future developments of the framework.

Block interface

```
2  // ------------------------------
3  // COMMON (Step = -1)
4  // ------------------------------
5  "LoadTransfer_DB"(I_AtLeftExit := "At left entry",
6                    I_AtRightExit := "At right entry",
7                    I_HighBox := "High sensor",
8                    I_Loaded := "Loaded",
9                    I_LowBox := "Low sensor",
10                   I_pallet := "Pallet sensor",
11                   Q_StartLoad => "Load",
12                   Q_TransfLefT => "Transf. left",
13                   Q_TransfRight => "Transf. right");
14 "Conveyor2_DB"(I_BoxExit := "At right exit",
15                I_Transfer := "At right entry",
16                Q_CV => "Conveyor2_DB".Q_CV,
17                Q_Start => "Conveyor right");
18 "Conveyor3_DB"(I_BoxExit := "At left exit",
19                I_Transfer := "At left entry",
20                Q_CV => "Conveyor3_DB".Q_CV,
21                Q_Start => "Conveyor left");
22 "Conveyor4_DB"(I_Button := "Manual Button",
23                I_ConvLeft := "Conveyor left",
24                I_PickDone := "PickAndPlace_DB".Q_Done,
25                I_Sensor := "Box Sensor",
26                Q_Start => "Conveyor left 2",
27                Q_StartPick => "Conveyor4_DB".Q_StartPick);

28 "PickAndPlace_DB"(IO_grab := "Q_Grab",
29                   IO_RotLeft := "Q_LeftRot",
30                   IO_RotRight := "Q_RightRot",
31                   I_detect := "I_ItemDetected",
32                   I_rot := "I_Rotating",
33                   I_Sensor := "Box Sensor",
34                   I_Start := "Conveyor4_DB".Q_StartPick,
35                   I_x := "I_XPos",
36                   I_z := "I_ZPos",
37                   Q_Done => "PickAndPlace_DB".Q_Done,
38                   Q_x => "Q_XSet",
39                   Q_z => "Q_ZSet");
40
41 // ------------------------------
42 // MAIN FSM
43 // ------------------------------
44 CASE #Step OF
45     0:
46         IF "FACTORY I/O (Running)" THEN
47             "Conveyor_DB"(I_pallet := "Pallet sensor",
48                           Q_Start => "Conveyor entry");
49         END_IF;
50
51 END_CASE;
```

Figure 4.6: Generated main control logic coordinating the industrial units [4]

Therefore, this manual operation is only required when starting the new project for the first time. Once saved, no additional manual operation will be required, even after updating the project itself using the same application.

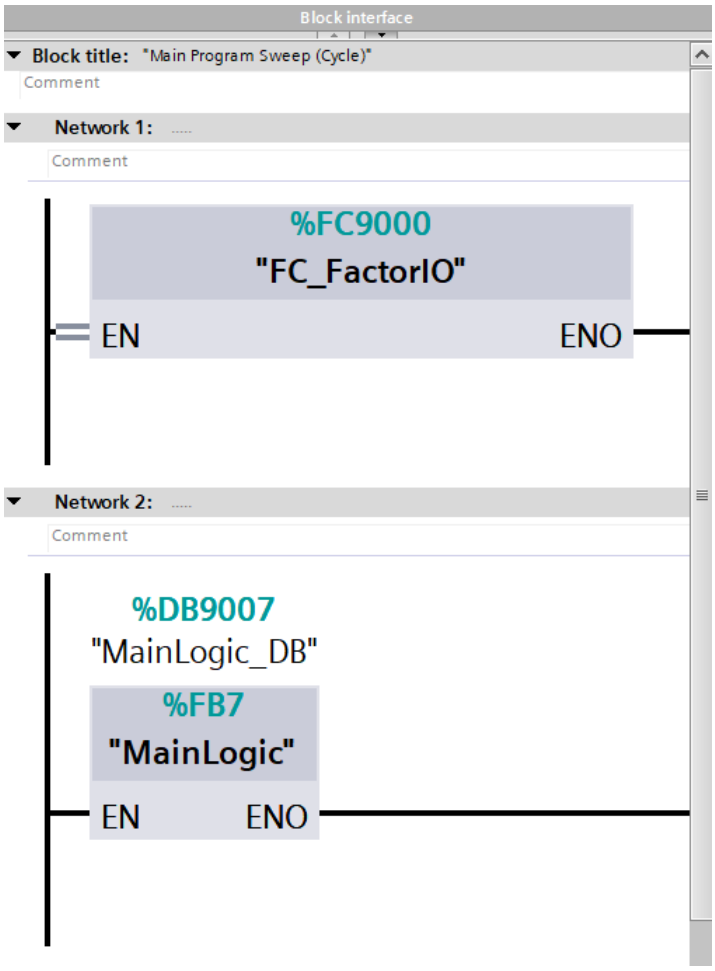


Figure 4.7: Manual insertion of the communication FC and the generated main FB into the two networks of the main program OB1 [4]

4.1.2 Incremental update of the PLC project

The project update pipeline is applied to the previously generated project in order to obtain the final configuration corresponding to the second industrial process. In this phase, the existing project is not recreated from scratch. The generator incrementally updates the previously generated software structure by modifying only the components affected by the new process specification. The overall structure described from the project tree and the hardware configuration remain the same of the first industrial process.

The first modification concerns the regeneration of the PLC tag table using the new Factory I/O driver configuration.

	Name	Data type	Address	Retain	Acces...	Writa...	Visibl...	Comment
1	Two-Axis Pick & Place 1 (Rotating)	Bool	%I0.0	☐	☑	☑	☑	Two-Axis Pick & Place 1 (Rotating)
2	Two-Axis Pick & Place 1 (Item Detected)	Bool	%I0.1	☐	☑	☑	☑	Two-Axis Pick & Place 1 (Item Detected)
3	Light Array Emitter 1 (Beam 6)	Bool	%I0.2	☐	☑	☑	☑	Light Array Emitter 1 (Beam 6)
4	Light Array Emitter 1 (Beam 7)	Bool	%I0.3	☐	☑	☑	☑	Light Array Emitter 1 (Beam 7)
5	Light Array Emitter 1 (Beam 8)	Bool	%I0.4	☐	☑	☑	☑	Light Array Emitter 1 (Beam 8)
6	Capacitive Sensor 1	Bool	%I0.5	☐	☑	☑	☑	Capacitive Sensor 1
7	FACTORY I/O (Running)	Bool	%I0.6	☐	☑	☑	☑	FACTORY I/O (Running)
8	Retroreflective Sensor 1	Bool	%I0.7	☐	☑	☑	☑	Retroreflective Sensor 1
9	Diffuse Sensor 1	Bool	%I1.0	☐	☑	☑	☑	Diffuse Sensor 1
10	Diffuse Sensor 2	Bool	%I1.1	☐	☑	☑	☑	Diffuse Sensor 2
11	Retroreflective Sensor 4	Bool	%I1.2	☐	☑	☑	☑	Retroreflective Sensor 4
12	Vision Sensor 1	Bool	%I1.3	☐	☑	☑	☑	Vision Sensor 1
13	Two-Axis Pick & Place 1 X Position (V)	Real	%ID30	☐	☑	☑	☑	Two-Axis Pick & Place 1 X Position (V)
14	Two-Axis Pick & Place 1 Z Position (V)	Real	%ID34	☐	☑	☑	☑	Two-Axis Pick & Place 1 Z Position (V)
15	Two-Axis Pick & Place 1 Rotate CW	Bool	%Q0.0	☐	☑	☑	☑	Two-Axis Pick & Place 1 Rotate CW
16	Two-Axis Pick & Place 1 Rotate CCW	Bool	%Q0.1	☐	☑	☑	☑	Two-Axis Pick & Place 1 Rotate CCW
17	Two-Axis Pick & Place 1 (Grab)	Bool	%Q0.2	☐	☑	☑	☑	Two-Axis Pick & Place 1 (Grab)
18	Chain Transfer 1 (+)	Bool	%Q0.3	☐	☑	☑	☑	Chain Transfer 1 (+)
19	Chain Transfer 1 (-)	Bool	%Q0.4	☐	☑	☑	☑	Chain Transfer 1 (-)
20	Chain Transfer 1 (Left)	Bool	%Q0.5	☐	☑	☑	☑	Chain Transfer 1 (Left)
21	Chain Transfer 1 (Right)	Bool	%Q0.6	☐	☑	☑	☑	Chain Transfer 1 (Right)
22	Roller Conveyor (4m) 1	Bool	%Q0.7	☐	☑	☑	☑	Roller Conveyor (4m) 1
23	Roller Conveyor (4m) 2	Bool	%Q1.0	☐	☑	☑	☑	Roller Conveyor (4m) 2
24	Roller Conveyor (4m) 3	Bool	%Q1.1	☐	☑	☑	☑	Roller Conveyor (4m) 3
25	Roller Conveyor (4m) 4	Bool	%Q1.2	☐	☑	☑	☑	Roller Conveyor (4m) 4
26	Two-Axis Pick & Place 1 X Set Point (V)	Real	%QD30	☐	☑	☑	☑	Two-Axis Pick & Place 1 X Set Point (V)
27	Two-Axis Pick & Place 1 Z Set Point (V)	Real	%QD34	☐	☑	☑	☑	Two-Axis Pick & Place 1 Z Set Point (V)
28	Digital Display 1	DInt	%QD38	☐	☑	☑	☑	Digital Display 1

Figure 4.8: Upsert of the PLC tag table generated from the driver configuration of the second industrial process [4]

Another relevant modification concerns the recycling conveyor logic. Figure 4.9 and Figure 4.10 show the different generated control logic before and after the update pipeline. In the first industrial process, there is a timer for the manual button that is responsible for activating Pick&Place. In the second industrial process, the manual button is removed and the only control over the activation of Pick&Place is performed on the resulting value of the vision sensor. The conveyor behaviour is modified to interact with the vision sensor and the new automatic material sorting mechanism implemented in the Factory I/O environment.

Block interface

```
1
2  // *** Code from Excel ***
3
4 CASE #Step OF
5      0:
6          // OUT
7          IF #I_ConvLeft AND NOT #I_Sensor THEN
8              #Q_Start := TRUE;
9          END_IF;
10
11         // TRANS
12         IF #I_Sensor THEN
13             #Step := 1;
14         END_IF;
15
16     1:
17         // OUT (no condition)
18         #Q_Start := FALSE;
19         #tMan(IN := #I_Sensor,
20               PT := T#3s);
21
22         // TRANS
23         IF #I_Button THEN
24             #Step := 2;
25         ELSIF #tMan.Q THEN
26             #Step := 3;
27         END_IF;
28
29     2:
30         // OUT (no condition)
31         #Q_Start := TRUE;
32         #tMan(IN := FALSE,
33               PT := T#3s);
34
35         // TRANS
36         IF NOT #I_Sensor THEN
37             #Step := 0;
38         END_IF;
39
40     3:
41         // OUT (no condition)
42         #Q_StartPick := TRUE;
43         #tMan(IN := FALSE,
44               PT := T#3s);
45
46         // TRANS
47         IF #I_PickDone THEN
48             #Step := 4;
49         END_IF;
50
51     4:
52         // OUT (no condition)
53         #Q_Start := TRUE;
54         #Q_StartPick := FALSE;
55
56         // TRANS
57         IF NOT #I_Sensor THEN
58             #Step := 0;
59         END_IF;
60
61 END_CASE;
```

Figure 4.9: Generated `Conveyor4` FB showing the SCL implementation of the state machine logic in the first industrial process [4]

Block interface

```
1
2  // *** Code from Excel ***
3
4  CASE #Step OF
5       0:
6            // OUT
7            IF #I_ConvLeft AND NOT #I_Sensor THEN
8                 #Q_Start := TRUE;
9            END_IF;
10
11           // TRANS
12           IF #I_Sensor THEN
13                #Step := 1;
14           END_IF;
15
16      1:
17           // OUT (no condition)
18           #Q_Start := FALSE;
19           #Q_StartPick := TRUE;
20
21           // TRANS
22           IF #I_PickDone THEN
23                #Step := 2;
24           END_IF;
25
26      2:
27           // OUT (no condition)
28           #Q_Start := TRUE;
29           #Q_StartPick := FALSE;
30
31           // TRANS
32           IF NOT #I_Sensor THEN
33                #Step := 0;
34           END_IF;
35
36 END_CASE;
```

Figure 4.10: Updated `Conveyor4` FB showing the SCL implementation of the state machine logic in the second industrial process [4]

Finally, the main control logic is updated as illustrated in Figure 4.11. As discussed in the previous chapter, the main control logic can be updated in two different ways. The update consists in modifications of the I/O mapping according to the structure of the second industrial process.

Block interface

```
1
2  // ------------------------------
3  // COMMON (Step = -1)
4  // ------------------------------
5  "LoadTransfer_DB"(I_AtLeftExit := "Diffuse Sensor 2",
6                    I_AtRightExit := "Diffuse Sensor 1",
7                    I_HighBox := "Light Array Emitter 1 (Beam 6)",
8                    I_Loaded := "Capacitive Sensor 1",
9                    I_LowBox := "Light Array Emitter 1 (Beam 7)",
10                   I_pallet := "Light Array Emitter 1 (Beam 8)",
11                   Q_StartLoad => "Chain Transfer 1 (+)",
12                   Q_TransfLefT => "Chain Transfer 1 (Left)",
13                   Q_TransfRight => "Chain Transfer 1 (Right)");
14 "Conveyor2_DB"(I_BoxExit := "Retroreflective Sensor 1",
15                 I_Transfer := "Diffuse Sensor 1",
16                 Q_CV => "Conveyor2_DB".Q_CV,
17                 Q_Start => "Roller Conveyor (4m) 1");
18 "Conveyor3_DB"(I_BoxExit := "Retroreflective Sensor 4",
19                 I_Transfer := "Diffuse Sensor 2",
20                 Q_CV => "Conveyor3_DB".Q_CV,
21                 Q_Start => "Roller Conveyor (4m) 2");
22 "Conveyor4_DB"(I_ConvLeft := "Roller Conveyor (4m) 2",
23                 I_PickDone := "PickAndPlace_DB".Q_Done,
24                 I_Sensor := "Vision Sensor 1",
25                 Q_Start => "Roller Conveyor (4m) 4",
26                 Q_StartPick => "Conveyor4_DB".Q_StartPick);
27 "PickAndPlace_DB"(IO_grab := "Two-Axis Pick & Place 1 (Grab)",
28                   IO_RotLeft := "Two-Axis Pick & Place 1 Rotate CCW",
29                   IO_RotRight := "Two-Axis Pick & Place 1 Rotate CW",
30                   I_detect := "Two-Axis Pick & Place 1 (Item Detected)",
31                   I_rot := "Two-Axis Pick & Place 1 (Rotating)",
32                   I_Sensor := "Vision Sensor 1",
33                   I_Start := "Conveyor4_DB".Q_StartPick,
34                   I_x := "Two-Axis Pick & Place 1 X Position (V)",
35                   I_z := "Two-Axis Pick & Place 1 Z Position (V)",
36                   Q_Done => "PickAndPlace_DB".Q_Done,
37                   Q_x => "Two-Axis Pick & Place 1 X Set Point (V)",
38                   Q_z => "Two-Axis Pick & Place 1 Z Set Point (V)");
39
40 // ------------------------------
41 // MAIN FSM
42 // ------------------------------
43 CASE #Step OF
44     0:
45         IF "FACTORY I/O (Running)" THEN
46             "Conveyor_DB"(I_pallet := "Light Array Emitter 1 (Beam 8)",
47                           Q_Start => "Roller Conveyor (4m) 3");
48         END_IF;
49
50 END_CASE;
```

Figure 4.11: Updated main control logic coordinating the industrial units [4]

4.1.3 Engineering effort comparison

Table 4.1: Comparison between manual PLC engineering workflow and automated generation using the proposed framework: approximated numerical values

Engineering Task	**Manual Workflow**	**Automated Framework**
Project creation time	2-3 hours	15-30 minutes
Update project time	1 hour	10 minutes
Remaining manual steps time		2 minutes
Tag table creation time	40 minutes	2 minutes
FBs and DBs creation time	1 hour	10 minutes
I/O mapping configuration	30 minutes	5-10 minutes
Tag and address mismatch risk	High possibility due to manual mapping	0% (Automatically aligned with Factory I/O mapping)
Compile success	Depends on correct manual configuration	100% (Successful in case studies)
Scalability of project structure	Manual replication of blocks, manual tags adding, manual I/O mapping adding	Automatic blocks, tags and mapping adding

The comparison presented in Table 4.1 highlights the benefits of the proposed framework compared to a traditional PLC engineering workflow. In a manual development process, engineers must sequentially configure hardware devices, create tag tables, instantiate function blocks and manually implement control logic. These activities require a considerable amount of engineering time and are prone to configuration inconsistencies. By contrast, the proposed framework automatically generates the majority of the engineering artefacts from structured configuration files. As a result, the time required to generate the PLC project is significantly reduced, and the number of manual configuration steps is limited. The evaluation also suggests that the automated approach improves the consistency between the simulation environment and the PLC implementation by deriving signal definitions directly from the Factory I/O driver configuration. This mechanism reduces the risk of addressing mismatches and naming inconsistencies that can occur during manual PLC configuration.

4.1.4 Consistency of generated engineering artefacts

Another aspect considered in the evaluation is the consistency between the generated PLC project and the external configuration sources used to describe the industrial process. In traditional PLC engineering workflows, the configuration of

signals, tag names and memory addresses is typically performed manually. This manual configuration step can introduce inconsistencies between the simulation environment, the PLC tag tables and the control logic implementation. Typical errors include incorrect addressing, mismatched signal names or incomplete mappings between sensors, actuators and program variables. In the proposed framework, the PLC tag tables are automatically generated from the Factory I/O driver configuration exported in XML format. This mechanism ensures that the signal names, data types and logical addresses used in the PLC program remain consistent with the configuration defined in the simulation environment. In addition, the generation pipeline automatically instantiates the required function blocks and their associated instance data blocks, using a deterministic naming convention derived from the configuration files. This approach guarantees that each industrial unit is consistently represented within the PLC project structure. The results presented in the previous sections confirm that the generated project preserves consistency across the different engineering artefacts. The automatically generated tag tables, block interfaces and instance data blocks maintain a coherent mapping between the simulation environment and the PLC implementation, reducing the likelihood of configuration errors typically associated with manual engineering workflows.

4.1.5 Scalability and structural manageability

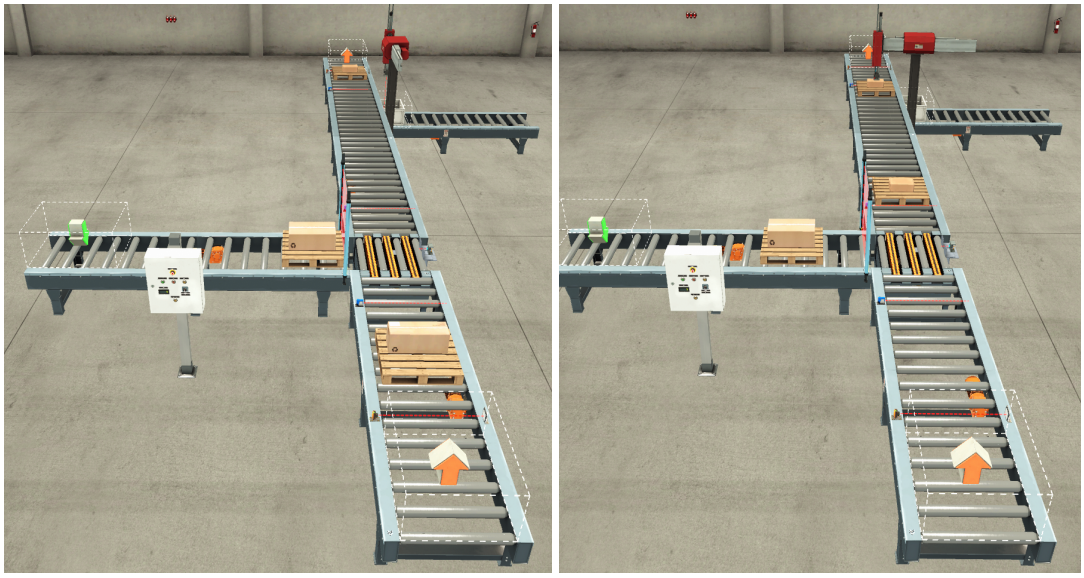
The scalability of the proposed framework is also considered in the evaluation. In industrial automation systems, the complexity of PLC projects typically increases with the number of controlled devices, sensors and actuators. As systems grow, maintaining a clear and manageable software architecture becomes increasingly challenging in traditional engineering workflows. The framework proposed in this thesis generates PLC software following a modular architecture in which each industrial unit is represented by a dedicated function block and an associated instance data block. This modular structure allows the behaviour of each process unit to remain encapsulated and independent from other parts of the automation system. When additional process units are introduced, the generation pipeline can automatically instantiates the corresponding blocks and updates the main control logic accordingly. This mechanism avoids manual duplication of control logic and helps maintain a structured project organisation inside the TIA Portal environment. Although the case studies considered in this thesis involve a limited number of industrial units, the modular generation strategy suggests that the framework can support the extension of the automation system without requiring major modifications to the existing project structure. Each additional unit results in the automatic creation of the required tags, blocks and instance databases while preserving the architectural consistency of the PLC project. In scenarios where the industrial system becomes

more complex and includes new types of process units, the extension of the framework can be achieved by modifying the configuration artefacts used as input for the generation pipeline. In particular, new process units can be described by adding dedicated configuration sheets in the Excel specification files and by extending the corresponding JSON configuration structures. If required, additional generation rules can also be implemented within the software application to support the new unit types. The modular design of the generation framework facilitates this extension process. Because the generation logic is organised into dedicated software modules, modifications can be implemented in specific parts of the code without requiring substantial changes to the entire application. This modular organisation supports the adaptation of the framework to different industrial scenarios while preserving the overall generation workflow. Nevertheless, the current implementation has been validated only on relatively simple industrial processes involving a limited number of units. While the proposed architecture is designed to support extension to more complex automation systems, further validation on larger industrial scenarios would be necessary to fully assess the scalability of the approach. For this reason, the proposed framework can be considered as an extensible solution for automated PLC project generation, capable of supporting the evolution of industrial automation systems through configuration-driven extensions and modular software design.

4.2 Validation through Factory I/O simulation

The generated PLC projects were finally validated through the execution of the corresponding industrial processes in Factory I/O simulation environment. This step allows the verification of the interaction between the generated control logic, the virtual sensors and actuators of the simulated industrial process, and the communication established through Siemens PLCsim. The simulation environment enables the execution of the complete automation cycle using the PLC program generated by the proposed framework. The results demonstrate that the generated control logic correctly interacts with the simulated devices and executes the intended process behaviour. For the first industrial process, the generated PLC project successfully controls the conveyor system, the LoadTransfer unit and the Pick&Place manipulator according to the specified control sequence. The observed system behaviour is consistent with the process specification used as input for the generation pipeline. After the execution of the update pipeline, the modified PLC project also correctly implements the second industrial process. The updated control logic interacts with the additional sensors and actuators introduced in the new simulation scenario, illustrating the capability of the framework to incrementally adapt an existing PLC

project to a modified process configuration.
Figure 4.12 illustrates the simulation of the first industrial process. The boxes are correctly sorted based on their height. Small boxes are transferred from the Load-Transfer unit on the left conveyor. The images show the two different operating conditions for Pick&Place unit. In Figure 4.12a is illustrated the case of manual button pressing before the timer end. The conveyor is reactivated and Pick&Place unit is not enabled. Figure 4.12b shows the case in which the manual button is not pressed, and after three seconds the Pick&Place unit is activated.



(a) Manual button pressed [4] (b) Pick&Place activation [4]

Figure 4.12: Simulation of the first industrial process [4]

Figure 4.13 provides a detailed view of the Pick&Place unit during the manipulation phase of the first industrial process. The images highlight the behaviour of the manipulator and the corresponding values of the sensors and actuators involved in the operation. In particular, the zoomed view shows the position variables and the activation of the control signals associated with the Pick&Place unit. The manipulator correctly detects the pallet and executes the gripping operation according to the control logic generated in the PLC program. The displayed sensor values confirm that the automated control logic correctly manages the motion and gripping phases of the manipulation task.

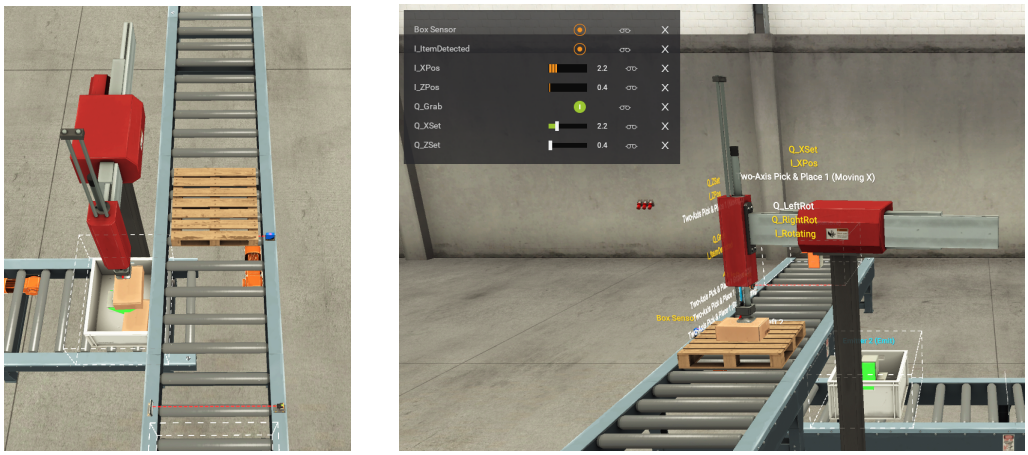
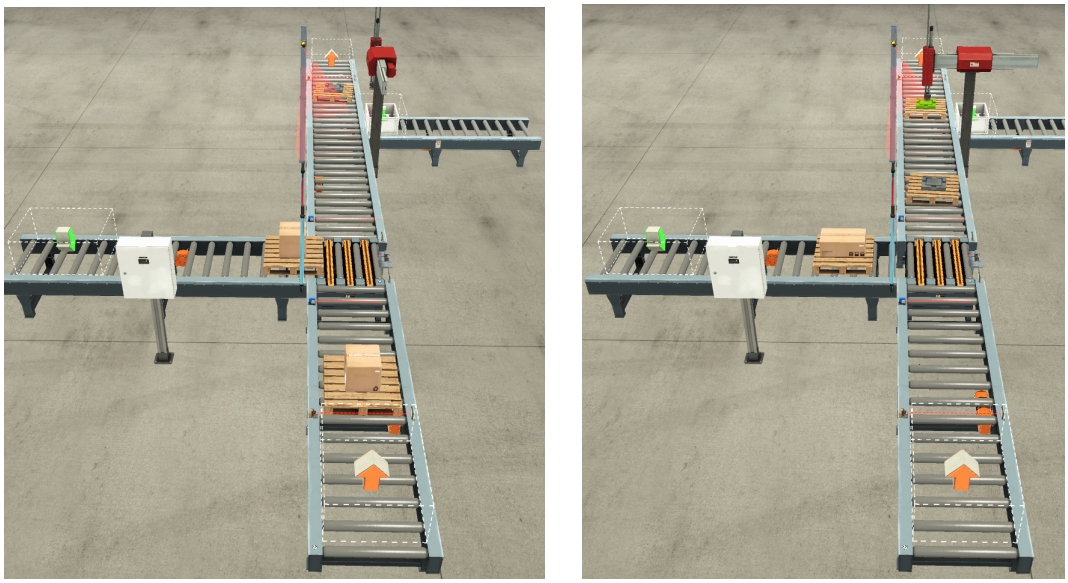


Figure 4.13: Detailed view of the Pick&Place unit during the manipulation phase of the first industrial process [4]

Figure 4.14 illustrates the simulation of the second industrial process. The boxes are correctly transferred on the right conveyor. Objects are transferred from the LoadTransfer unit on the left conveyor. The images show the two different operating conditions for Pick&Place unit. In Figure 4.14a is illustrated the case of metallic object. The vision sensor gives zero value, the conveyor is moving and the Pick&Place is not enabled. Figure 4.14b shows the case in which the object is non-metallic. In this case the vision sensor returns a value equal to one, the conveyor stops and Pick&Place unit is activated.



(a) Case of metallic object [4]

(b) Case of non-metallic object [4]

Figure 4.14: Simulation of the second industrial process [4]

Figure 4.15 shows a detailed view of the Pick&Place manipulation phase in the second industrial process. The images illustrate the interaction between the vision sensor and the manipulator control logic. When a non-metallic object is detected, the conveyor stops and the Pick&Place unit is activated in order to pick the object and place it into the dedicated container. The zoomed view highlights the values of the sensor signals and the actuator commands controlling the manipulator motion. In addition, the `stackable box` shows the correctness of the sorted objects resulting from the automated classification process implemented in the control logic.

Figure 4.15: Detailed view of the Pick&Place unit during the manipulation phase of the second industrial process [4]

These observations indicate that the control logic generated by the proposed framework correctly implements the intended process behaviour in both industrial scenarios.

4.3 Limitations of the proposed approach

Although the proposed framework demonstrates promising results for automated PLC project generation, several limitations must be acknowledged. First, the validation of the framework has been performed primarily using the Factory I/O simulation environment combined with Siemens PLCsim. While this setup provides a realistic virtual commissioning environment, further validation on real industrial hardware would be necessary to fully assess the robustness of the approach in production scenarios. Second, the case studies considered in this thesis involve relatively simple industrial processes with a limited number of automation units. Although the modular architecture of the framework is designed to support larger systems, additional experimentation on more complex industrial scenarios would be required to fully evaluate its scalability. Another limitation concerns the partial manual step

that remains in the current workflow. In the present implementation, the communication function and the generated main logic block must be manually inserted into the OB1 program block. While this operation is minimal, it indicates that the workflow is not yet fully end-to-end automated. Finally, the current framework does not yet include a formal automated testing layer capable of verifying the correctness of the generated control logic. Future work could integrate automated validation mechanisms to further improve the reliability of the generation process. Despite these limitations, the results obtained in the case studies indicate that the proposed approach represents a promising step toward configuration-driven PLC engineering and automated project generation within industrial automation environments.

Chapter 5

Conclusions and future developments

5.1 Conclusions

This thesis introduces a configuration-driven transformation framework for the automated creation and update of PLC projects in TIA Portal, through a modular C# application interacting with *TIA Portal Openness*. The proposed methodology formalizes PLC engineering as a deterministic transformation process structured into defined layers:

- Industrial process modelling in Factory I/O.
- Structured configuration layer including Excel and JSON files.
- Interactive orchestration layer by using a CLI-driven execution.
- C# coding as transformation engine.
- *TIA Portal Openness* as programmable engineering interface.
- Generated or updated directly PLC project in TIA Portal.

This layered architecture separates specification, transformation logic and engineering execution, enabling a structured automation workflow designed to support the lifecycle of a PLC project in TIA Portal, from creation to updating. The implemented C# application acts as a transformation orchestrator. It parses configuration artifacts, merges user-driven specifications and activates either a project creation or update pipeline. The engineering actions are executed through *TIA Portal Openness*, which serves as a programmable API layer abstracted from the transformation logic.

From a technical perspective, the evaluation presented in this thesis highlights the following results:

- Automated generation of PLC tag tables directly from the simulation environment configuration implemented in Factory I/O.
- Controlled instantiation of PLC rack and program blocks.
- Selective regeneration and update of projects components.
- Preservation of existing project elements during incremental updates.
- Standardized naming conventions and structural consistency across projects.

The transformation pipeline reduces manual configuration effort and helps minimize potential inconsistencies between simulation and PLC implementation. By importing I/O definitions directly from the simulation environment, the framework eliminates redundant configuration steps and significantly reduces the probability of addressing errors.
The modular service-oriented design of the C# application ensures:

- Clear separation of concerns.
- Extensibility toward additional generation features.
- Maintainability of transformation logic.
- Support for scalable extension across different project models.

Furthermore, the use of a guided CLI introduces controlled variability and customization in execution and implementation steps. The system supports human-supervised orchestration while maintaining deterministic generation rules. Overall, the results demonstrate that PLC project engineering can be formalised as a programmable and configuration-driven transformation rather than a purely manual design activity.

Achieved advantages

The proposed solution provides the following advantages:

- **Structural Standardization**

  The framework enforces consistent naming conventions, uniform project organization and repeatable generation rules. This reduces architectural drift across projects and supports long-term maintainability.

- **Reduction of human-induced errors**

  Manual operations are common sources of error in complex automation projects. Automated generation and update of projects, by using modular C# application, structured configuration artifacts and CLI-driven specifications, can help reduce this risk.

- **Replicability and portability**

  Projects and their specific blocks can be replicated or adapted by modifying configuration files rather than rewriting logic. This enables reuse across different projects with different complexity structure, replicating blocks on a large scale and easily integrating new blocks.

- **Virtual commissioning enablement**

  By interacting simulation modelling with programmable PLC generation, the framework supports early validation in virtual simulation environments. This reduces commissioning time and improves reliability prior to physical deployment.

- **Modularity and extensibility**

  The service-based architecture allows new generation features, update strategies, or configuration sources to be integrated without refactoring the entire system. CLI-driven configuration offers tailor-made functionality for specific parts of the system, allowing the user to update logic and integrate new systems. This project lays the foundations for the automatic development of complex industrial processes, with multiple blocks and shared logic.

Final considerations

The work presented in this thesis shows that PLC engineering workflows can be formalized and automated through structured transformation pipelines driven by configuration artifacts and programmable engineering interfaces. By leveraging a model-based configuration approach together with the engineering capabilities offered by *TIA Portal Openness*, the proposed framework enables the automated generation and lifecycle management of Siemens PLC projects. The integration with simulation environments such as Factory I/O further supports testing and validation of the generated control logic within a virtual industrial process. By abstracting engineering logic from manual graphical programming and encapsulating it into deterministic generation services, the framework contributes to the digitalization and

standardization of industrial automation engineering practices. Overall, the proposed approach demonstrates how model-based configuration, programmable engineering APIs, and simulation environments can be combined to support automated and configuration-driven PLC project development while maintaining modularity, scalability, and engineering consistency. In conclusion, although the validation presented in this thesis is limited to the case studies considered, the results suggest that the proposed framework represents a promising step towards PLC engineering based on configuration and automated project generation in industrial automation environments.

5.2 Future developments

Although the proposed framework achieves structured and automated PLC project generation while supporting the entire lifecycle management of industrial automation projects, several technical extensions could further enhance its capabilities and scalability.

- **Automated validation and testing layer**

  An extension would consist of implementing an automated testing and validation module capable of verifying the consistency and correctness of the generated project. Such a module could perform:

  - Verifying tag consistency across configuration files and PLC structures.
  - Checking logical dependencies between generated blocks.
  - Validating naming compliance.
  - Generating structured diagnostic and validation reports.

  This would introduce a verification layer between transformation and deployment phases, improving reliability and reducing commissioning errors.

- **Integration with advanced digital twin environments**

  The framework could be extended to integrate with advanced industrial simulation and digital twin platforms such as 3D factory simulators and robotic simulation environments. By enabling seamless interoperability between mechanical design tools, simulation platforms, and PLC generation workflows, the framework could support the creation and validation of industrial processes within virtual environments before physical deployment. Maintaining synchronization between the simulation models and the automatically generated PLC control logic would allow more effective virtual commissioning,

enabling engineers to test and validate system behaviour in a digital environment and significantly reducing commissioning time.

- **AI-supported configuration assistance**

  An intelligent assistant layer could support engineers during project configuration and commissioning activities, providing:

  - Automatic parameter configuration suggestions.
  - Control logic recommendations based on predefined industrial templates.
  - Anomaly detection in configuration files.
  - Guided update and maintenance recommendations during system evolution.

  This would enhance decision support during project creation and maintenance phases and could assist engineers during testing, parameter tuning and commissioning.

- **Template-Based industrial module libraries**

  The configuration part could include the development of reusable libraries of industrial automation modules. These libraries would contain standardized templates representing typical industrial components or machine units. Through configuration rules, these modules could be instantiated dynamically within the generated PLC project, allowing faster system design and improving scalability when dealing with complex industrial plants composed of multiple coordinated subsystems.

- **Extension toward more complex industrial process architectures**

  A further development could consist in the extension of the framework to support the generation of more complex industrial processes composed of a larger variety of automation modules and subsystems. The current implementation focuses on a limited set of process units, such as conveyors, load transfer stations, and pick-and-place modules. Future extensions could introduce additional configurable modules representing a wider range of industrial operations and machine functionalities. By expanding the available modular components and improving the configuration structure, the framework could support the automatic generation of more articulated automation architectures composed of multiple interacting subsystems, shared control logic, and larger sets of inputs and outputs. This evolution would allow the framework to be applied to a broader class of industrial scenarios, enabling the development of complex automation systems while maintaining the advantages of modular

design, configuration-driven generation, and reduced manual programming effort.

List of Acronyms

AML AutomationML. 9, 16, 17

API Application Programming Interface. 9, 15, 16, 44–46, 75, 78, 83, 91, 92, 102

CAx Computer-Aided Technologies. 16

CLI Command-Line Interface. 44–48, 50, 52, 75–77

CSV Comma-Separated Values. 14

DSL Domain-Specific Language. 14

FB Function Block. 1, 3, 4, 6, 9, 14, 15, 18, 24, 31, 33, 38, 40–42, 46–53, 55, 61, 99

FBD Function Block Diagram. 6, 7

FC Function. 6, 34, 38, 46, 61

HMI Human–Machine Interface. 8, 15, 16

I/O Input / Output. 1–4, 6, 9, 14–18, 22–25, 30–32, 34, 35, 38–40, 42, 45–47, 50–57, 64, 66, 68–70, 73, 75–77, 90, 91

IEC International Electrotechnical Commission. 2, 3, 6–9, 11, 18–20

IFC Industrial Function Concept. 3, 11, 12, 17–19

instance DB Instance Data Block. 4, 6, 16, 18, 38, 40, 41, 46–49, 52, 55, 58, 99

JSON JavaScript Object Notation. 17, 23, 34, 39–42, 44, 46, 47, 50–53, 70, 75, 99, 100

LAD Ladder Diagram. 6, 7, 33

OB Organization Block. 6, 61, 74

PCG PLC Code Generation. 13, 18

PLC Programmable Logic Controller. 1–4, 6–25, 31–40, 42, 44–46, 50, 54, 55, 57, 64, 68–71, 73–79, 83–85, 90–92, 97, 99, 101, 102

SCL Structured Control Language. 6, 41, 47, 58, 86, 99

SIPN Signal Interpreted Petri Net. 11, 17

SysML Systems Modeling Language. 7

TIA totally Integrated Automation. 2–4, 6, 9, 13, 15, 16, 18, 20–25, 30–32, 34–36, 38–41, 44–47, 49–51, 54, 55, 69, 75, 77, 83, 86, 88, 90–92, 102

UML Unified Modeling Language. 7

XML eXtensible Markup Language. 3, 4, 9, 10, 14, 17, 30, 32, 34, 39, 40, 46, 51, 57, 69, 90, 92

Chapter 6

Appendix A

6.1 Configuration file overview

This appendix provides a detailed description of the Excel configuration file used by the framework to describe the hardware configuration and the automation logic of the industrial process. The Excel file acts as a high-level specification artifact that is interpreted by the C# generator to automatically construct the corresponding Programmable Logic Controller (PLC) project in totally Integrated Automation (TIA) Portal through the *TIA Portal Openness* Application Programming Interface (API). The configuration file is organized as a multi-sheet Excel document. Each sheet describes a specific aspect of the automation system, including hardware configuration, tag definitions and state machine logic of the implemented function blocks. Table 3.1 in Section 3.3.1 summarizes the general structure of the configuration file and the role of each sheet.

6.2 Hardware configuration sheets

The first two sheets of the Excel file define the hardware configuration of the PLC system. Each row of the sheet represents a hardware element that will be created within the TIA Portal project. The meaning of the columns used in the hardware configuration sheets is summarized below.

Table 6.1: Columns used in hardware configuration sheets

Column	Description
deviceName	Name assigned to the device instance in the generated PLC project
deviceItemName	Device type used in the hardware configuration
typeIdentifier	Siemens order number identifying the specific hardware component including its version

Examples of hardware configuration entries are already presented in the tables reported in Section 3.3.1.

6.3 Tag configuration sheet

The third sheet of the Excel file is dedicated to the definition of PLC tags. Each row represents a variable that will be created in the PLC tag table of the generated project. The structure of the tag configuration sheet is described by the following columns.

Table 6.2: Columns used in tag configuration sheet

Column	Description
Name	Name of the PLC tag
DataType	Data type associated with the variable
LogicalAddress	PLC memory address assigned to the variable
Comment	Description of the tag functionality

6.4 State machine configuration sheets

The remaining sheets of the Excel file describe the behaviour of the industrial units through state machine specifications. Each sheet corresponds to the configuration of a specific function block used in the automation system. The state machines describe the operational logic of units such as conveyors, Pick&Place systems and LoadTransfer mechanisms. Each row of the configuration sheet represents a transition rule of the state machine. The structure used to describe the state machine logic is composed of five columns.

Table 6.3: Columns used in state machine configuration

Column	Description
step	Identifier of the current state
OutCondition	Input condition required to activate the state actions
actions	Outputs activated during the execution of the state
TransCondition	Logical condition that triggers the transition to the next state
nextStep	Identifier of the next state reached after the transition

	A	B	C	D	E
1	step	OutCondition	actions	TransCondition	nextStep
2	0	I_ConvLeft AND NOT I_Sensor	Q_Start:=TRUE	I_Sensor	1
3	1		Q_Start:=FALSE; tMan(IN := #I_Sensor, PT := T#3s)	I_Button	2
4				tMan.Q	3
5	2		Q_Start:=TRUE; tMan(IN := FALSE, PT := T#3s)	NOT I_Sensor	0
6	3		Q_StartPick:=TRUE; tMan(IN := FALSE, PT := T#3s)	I_PickDone	4
7	4		Q_Start:=TRUE; Q_StartPick:=FALSE	NOT I_Sensor	0

Figure 6.1: Example of state machine configuration in the Excel file [4]

6.5 Main logic configuration

The last sheet of the Excel configuration file is used to define the structure of the main PLC logic. Unlike the other sheets, which describe the state machines of individual function blocks, this sheet specifies the execution sequence of the function block instances within the main program. An example of the Main sheet is shown in Figure 6.2. In this case, the sheet is used to generate the sequence of calls that will be inserted into the main logic block of the PLC project.

	A	B	C	D	E
1	step	OutCondition	actions	TransCondition	nextStepText
2	0	"FACTORY I/O (Running)"	CALL Conveyor_DB		
3	-1		CALL LoadTransfer_DB		
4	-1		CALL Conveyor2_DB		
5	-1		CALL Conveyor3_DB		
6	-1		CALL Conveyor4_DB		
7	-1		CALL PickAndPlace_DB		

Figure 6.2: Example of main logic configuration in the Excel file [4]

The first row defines the column names, consistently with the structure adopted for the other sheets. However, in the Main sheet, the semantic role of the columns is partially adapted to represent the execution flow of the main control program. In particular, the *actions* column contains the calls to the function block instances that must be executed cyclically in the main program. The first row of the configuration uses the condition `"FACTORY I/O (Running)"` in the *OutCondition* column to identify the runtime condition under which the main execution logic is enabled. The value in the *step* column is set to `0` for this entry, while the following rows use the value `-1` to indicate that they do not represent state transitions, but sequential function block calls belonging to the same execution structure. Therefore, the Main sheet does not define a state machine in the strict sense. Instead, it provides a structured representation of the cyclic call order of the generated function blocks, allowing the generator to automatically construct the corresponding main program logic in the TIA Portal project.

6.6 Naming conventions and logical expressions

The Excel configuration files follow a set of naming conventions that simplify the automatic interpretation of variables and expressions by the source-code generator. These conventions are consistently adopted across all function block configuration sheets and are used by the generator to infer the interface and internal variables of the generated Structured Control Language (SCL) function blocks. Variable classification is primarily based on name prefixes. The conventions used by the generator are summarized in Table 6.4.

Table 6.4: Naming conventions used by the function block generator

Prefix / Pattern	**Interpretation in the generator**
`I_`	Input variable, inserted in the `VAR_INPUT` section
`Q_`	Output variable, inserted in the `VAR_OUTPUT` section
`IO_`	Input/output variable, inserted in the `VAR_IN_OUT` section
`tmp_` or `_tmp`	Temporary variable, inserted in the `VAR_TEMP` section
`t...` used with `.Q` or `.ET`	Timer instance automatically interpreted as a static variable of type `TON_TIME`
`_addTrig`, `_addExit`, `_exitTrig`, or names ending in `Trig`	Trigger instance automatically interpreted as a static variable of type `R_TRIG`
No specific prefix	Internal static variable, inserted in the `VAR` section

The generator scans the expressions contained in the Excel rows and automatically classifies the encountered identifiers according to these conventions. Variables prefixed with `I_`, `Q_`, and `IO_` are used to build the external interface of the generated function block, while unnamed internal variables are treated as static variables. Temporary variables are recognized through the prefixes `_tmp`. These variables are placed in the temporary memory section of the generated function block and are intended for intermediate computations. Trigger variables are identified either through specific reserved names, such as `_addTrig`, `_addExit`, and `_exitTrig`, or through identifiers ending in `Trig`. These variables are automatically declared as instances of the `R_TRIG` function block. Timer variables are recognized when an identifier starting with `t` is accessed through members such as `.Q` or `.ET`. In this case, the generator declares the corresponding identifier as a static variable of type `TON_TIME`. Logical expressions used in the transition conditions follow standard boolean operators.

Chapter 7

Appendix B

7.1 Example of JSON configuration file

The JSON configuration file is used to centralize the execution parameters of the generator in a compact and easily editable format. This approach simplifies the configuration of the framework and avoids hard-coded parameters inside the source code. The file includes information related to the installed TIA Portal version, the paths of the required input files, the sheet mapping used for Excel parsing, and the default folders used during project generation. Listing 7.1 reports an example of the JSON configuration file used in the framework.

Listing 7.1: Example of JSON configuration file used by the generator

```
{
  "TiaVersion": "19.0",
  "ExcelFile": "\\\\vmware-host\\Shared Folders\\VSC Workspace\\IOfile.
     xlsx",
  "CpuSheet": 1,
  "IoSheet": 2,
  "TagSheet": 3,
  "PickAndPlaceSheet": 5,
  "ConveyorInfeedSheet" : 8,
  "LoadTransferSheet" : 7,
  "ConveyorDistributionSheet" : 6,
  "ConveyorRecyclingSheet" : 9,
  "TemplateProject": "C:\\Users\\tia\\Desktop\\PLCSIM-MyNewProject_V19
     \\PLCSIM-MyNewProject_V19.ap19",
  "TemplateBlockName": "MHJ-PLC-Lab-Function-S71200",
  "TemplateXmlPath": "\\\\vmware-host\\Shared Folders\\VSC Workspace\\
     FC_FactorIO.xml",
  "TagsXmlPath": "\\\\vmware-host\\Shared Folders\\VSC Workspace\\
```

```
    Tags_NewProcess_Siemens S7-PLCSIM_2026-01-19-11-45-07.xml",
  "ImportedBlockNumber": 9000,
  "ImportedBlockName": "FC_FactorIO",
  "DefaultProjectFolder": "C:\\Users\\tia\\Desktop"
}
```

The use of a JSON-based configuration file improves the maintainability of the framework, since execution parameters can be modified without changing the application source code. This also facilitates the adaptation of the workflow to different project instances and execution environments.

Chapter 8

Appendix C

8.1 Transformation pipeline overview

The algorithms reported in this appendix correspond to the internal logic implemented by the generator service modules described in the methodology chapter. Each algorithm abstracts the behaviour of the corresponding C# class used within the generation framework. The C# generator implements a transformation pipeline that converts the configuration artifacts into PLC project components within the TIA Portal engineering environment. The generator is structured as a set of specialised service classes that perform the different stages of the generation workflow. The main components involved in this transformation process are summarized in Table 8.1.

Table 8.1: Main generator service components

Class	**Role in the generation process**
`HardwareFlow`	Creation and configuration of PLC hardware devices and Input / Output (I/O) modules
`TagsFlow`	Generation of PLC tag tables from the Factory I/O eXtensible Markup Language (XML) driver configuration
`FunctionBlocksFlow`	Generation of function blocks and instance data blocks from Excel specifications
`MainLogicFlow`	Construction of the main PLC control logic and instance mapping
`BuildFlow`	Compilation and saving of the generated project

8.2 Hardware configuration generation

The hardware configuration of the PLC project is generated by the `HardwareFlow` component. This module reads the hardware specifications from the Excel configuration file and creates the corresponding devices inside the TIA Portal project through the Openness API.
The hardware generation procedure follows the algorithm reported in Algorithm 2.

Algorithm 2 Hardware generation from Excel configuration

1. Load Excel configuration file
2. Invoke `HardwareFlow.cs` module
3. Read CPU configuration sheet
4. For each row in CPU sheet:
   (a) Validate configuration fields
   (b) Create PLC device using TIA Portal Openness API
   (c) Retrieve CPU device item
   (d) Retrieve associated PLC software container
5. Set CPU protection access level to FULL ACCESS
6. If I/O modules are defined:
   (a) Locate hardware rack
   (b) For each module in I/O configuration sheet:
      i. Validate module configuration
      ii. Insert module into rack slot

The generator retrieves the CPU device item and accesses the corresponding PLC software container in order to enable further software generation steps such as tag creation and block generation.

8.3 Tag table generation from XML driver

The tag table of the PLC project is automatically generated from the driver configuration exported from the Factory I/O simulation environment. The driver con-

figuration is exported as an XML file containing the list of sensors and actuators associated with the industrial process.
The `TagsFlow` component parses this XML file and creates the corresponding tags in the PLC project through the TIA Portal Openness API.
The generation procedure is described in Algorithm 3.

Algorithm 3 Tag table generation from XML driver

1. Load Factory I/O XML driver configuration
2. Invoke `TagsFlow.cs` module
3. Create or retrieve tag table `IO_Tags`
4. Retrieve all existing PLC tag tables
5. Build a set of already used I/O addresses
6. For each `Tag` element in XML file:
   (a) Extract tag name
   (b) Extract data type
   (c) Extract logical address
   (d) If tag name is empty → skip
   (e) If logical address is missing → skip
   (f) If logical address is not an input/output address → skip
   (g) If address is already used → skip
   (h) If tag with same name already exists → skip
   (i) Create PLC tag
   (j) Assign datatype
   (k) Assign logical address
   (l) Assign comment

8.4 Conflict detection and consistency rules

During the generation process, the framework performs several consistency checks in order to avoid configuration conflicts within the PLC project.

These checks are implemented in the generator service and ensure that the generated project remains structurally valid and consistent with the configuration artifacts. The following consistency rules are applied during tag generation:

- Duplicate tag names are not allowed within the same tag table.
- Logical addresses must correspond to valid PLC input or output addresses.
- If an address is already assigned to an existing tag, the generator skips the creation of the new tag in order to prevent address conflicts.
- Tags without name or logical address attributes are ignored during parsing.

These validation rules ensure that the generated PLC configuration remains consistent with the addressing scheme defined in the Factory I/O driver configuration.

8.5 Function block generation procedure

This appendix describes the procedure used by the generator to automatically create and update PLC function blocks from the Excel configuration file. The generation of the control logic is implemented through a coordinated interaction between the classes responsible for orchestration, Excel parsing, source construction, and SCL logic generation. The complete workflow starts from the selection of the industrial unit to be generated, continues with the parsing of the corresponding Excel sheet, and ends with the generation of a TIA Portal function block and its associated instance data block.

Table 8.2: Main classes involved in function block generation

Class	**Role in the generation workflow**
`FunctionBlocksFlow`	Orchestrates the creation or update of PLC function blocks and instance DBs
`IOimporter.LoadFromXlsx`	Reads the Excel sheet associated with the selected industrial unit and returns the corresponding state-machine rows
`FbSourceBuilder`	Builds the complete SCL source of the function block, including interface and body
`SclGenerator_3`	Generates the state-machine SCL logic from the parsed Excel rows

Algorithm 4 Function block generation from Excel configuration

1. Invoke `FunctionBlocksFlow`
2. Ask the user which industrial unit must be managed:
   - PickAndPlace
   - Conveyor
   - LoadTransfer
3. If the selected unit is a Conveyor, ask the user to choose the conveyor logic variant:
   - Infeed
   - Distribution
   - Recycling
4. Determine the execution mode:
   - `Create`: create a new FB with the next free index
   - `Upsert`: create a new FB or update an existing one
5. Resolve the corresponding Excel sheet index and SCL source file
6. Execute `IOimporter.LoadFromXlsx` to read the state-machine rows from the selected sheet
7. Execute `FbSourceBuilder.BuildFbSource` to generate the complete FB source:
   - analyse variables used in conditions and actions
   - classify variables into interface and internal sections
   - execute `SclGenerator_3.GenerateStepLogic`
   - build the final SCL source file
8. Save the generated SCL source to a temporary `.scl` file
9. Import the SCL file into TIA Portal as an external source
10. Generate the corresponding FB from the imported source
11. If an old FB with the same name exists and execution mode is `Upsert`, delete the old FB before regeneration
12. Generate the associated instance DB:
    - derive the DB name from the FB name
    - assign the next free DB number
    - create the instance DB

8.5.1 Interface generation rules

The function block interface is generated automatically by analysing the identifiers appearing in the Excel state-machine specification. The class `FbSourceBuilder` scans the conditions, actions, and transition expressions contained in the parsed rows and classifies the detected identifiers according to predefined naming conventions. The main classification rules are the following:

- variables prefixed with `I_` are inserted in the `VAR_INPUT` section;
- variables prefixed with `Q_` are inserted in the `VAR_OUTPUT` section;
- variables prefixed with `IO_` are inserted in the `VAR_IN_OUT` section;
- variables identified as temporary are inserted in the `VAR_TEMP` section;
- all remaining internal variables are inserted in the `VAR` section;
- identifiers corresponding to timer instances are automatically declared as `TON_TIME`;
- identifiers corresponding to trigger instances are automatically declared as `R_TRIG`.

This mechanism guarantees consistency between the expressions written in the Excel configuration and the interface of the generated function block.

8.5.2 Generation of the state-machine logic

Once the Excel rows have been parsed and the variables have been classified, the class `SclGenerator_3` generates the internal SCL logic of the function block. The generated logic follows a state-machine structure based on the variable `Step`. Rows with `Step = -1` are interpreted as common logic valid independently of the current state, while the remaining rows are grouped according to the current state identifier.

For each state, the generator produces:

- the output logic associated with the current state,
- the transition conditions toward the next states,
- the corresponding update of the `Step` variable.

The resulting SCL body is inserted into a `CASE Step OF` structure, ensuring a deterministic implementation of the state machine described in the Excel sheet.

8.5.3 Create and upsert strategy

The function block generation procedure supports both project creation and project update. In `Create` mode, the framework always generates a new FB by using the next available index associated with the selected unit type. This avoids naming conflicts and allows multiple units of the same type to coexist within the same project. In `Upsert` mode, the user can either create a new block or update an already existing one. In the second case, the previous FB is deleted and regenerated from the updated Excel specification. The associated instance DB is also deleted and recreated in order to preserve consistency between the block interface and its memory structure. This strategy allows the framework to support the controlled evolution of the PLC project without requiring a complete reconstruction of the software architecture.

Chapter 9

Appendix D

9.1 Main logic mapping and generation procedure

This appendix describes the procedure used by the framework to generate the main PLC control logic from the Excel configuration and the interactive mapping wizard. Unlike the generation of the individual function blocks, the construction of the main logic is based on a semi-automatic strategy: the call structure is derived from the Excel specification, while the assignment of actual variables to the FB interface parameters is completed through user-guided interaction. The main logic generation workflow is implemented through the coordinated interaction of the classes listed in Table 9.1.

Table 9.1: Main classes involved in main logic generation

Class	**Role in the generation workflow**
`MainLogicFlow`	Orchestrates the creation or update of the main logic block
`MainMappingWizard`	Interactively builds or updates the parameter mapping between FB interface variables and PLC tags, constants, or expressions
`MainSourceBuilder`	Generates the SCL source code of the main logic block from the Excel rows and the stored mapping configuration
`MainMapping`, `CallMapping`, `Binding`	Data structures used to store the mapping configuration in JSON format

Algorithm 5 Main logic mapping and generation procedure

1. Invoke `MainLogicFlow`
2. Load the Main Excel sheet from the configured sheet index
3. Parse the rows of the Main sheet
4. Invoke `MainMappingWizard` to ensure the mapping configuration:
   (a) retrieve the list of available tags from `IO_Tags`
   (b) detect all `CALL <Instance>` statements in the Main Excel rows
   (c) for each detected instance:
      i. identify the corresponding instance DB
      ii. derive the associated FB type
      iii. export the FB to XML
      iv. parse the interface parameters from the XML file
      v. classify parameters as `IN`, `OUT`, or `INOUT`
      vi. prompt the user to assign each parameter to:
         - a PLC tag,
         - a constant,
         - an expression,
         - or leave it temporarily unbound
   (d) save the resulting mapping into a JSON file
5. Invoke `MainSourceBuilder` to generate the SCL source of the main logic:
   (a) read the Excel rows
   (b) read the stored mapping JSON file
   (c) replace each `CALL <Instance>` instruction with the corresponding SCL call
   (d) generate the main logic function block source
6. Save the generated source to a temporary `.scl` file
7. Import the SCL file into TIA Portal
8. Create or update the `MainLogic` block
9. If required, generate the associated instance DB

9.1.1 Interactive mapping strategy

The mapping between the function block interface variables and the actual PLC variables is created through an interactive wizard implemented by the class `MainMappingWizard`. The wizard first retrieves the available signal names from the PLC tag table `IO_Tags`. It then scans the Main Excel sheet and detects all occurrences of statements of the form `CALL <InstanceName>`. Each detected call is interpreted as a request to execute the corresponding FB instance inside the main program. For each detected instance, the generator locates the corresponding Instance Data Block (instance DB) in the PLC project and derives the name of the associated Function Block (FB) type. The FB is then exported as an XML file and its interface is parsed in order to retrieve the declared `IN`, `OUT`, and `INOUT` parameters.
For each parameter, the user is asked to define a binding. The supported binding types are:

- **Tag**, corresponding to a variable selected from the PLC tag table,
- **Constant**, such as `TRUE`, `5`, or `T#3s`,
- **Expression**, such as a logical expression over tags or internal values.

The resulting bindings are stored in a JavaScript Object Notation (JSON) file so that they can be reused during subsequent executions, especially in update mode.

9.1.2 Main logic source generation

Once the mapping has been defined, the class `MainSourceBuilder.cs` generates the SCL source code of the `MainLogic.cs` function block. The generator reads the rows of the Main Excel sheet and separates the rows associated with common logic from those associated with the step-based structure of the main finite state machine. For each row, the action expressions are processed and every occurrence of `CALL <Instance>` is replaced by a fully expanded SCL call. The final SCL call is generated according to the stored mapping configuration. In particular:

- input and in-out parameters are emitted using the operator `:=`,
- output parameters are emitted using the operator `=>`,
- tags are emitted as quoted PLC tag names,
- constants and expressions are inserted directly into the generated call.

This mechanism makes it possible to automatically generate the complete main control logic while preserving user-defined control over the actual variable bindings.

9.1.3 Create and update behaviour

The generation of the main logic supports both project creation and project update. In project creation mode, the `MainLogic` block is generated from scratch and inserted into the PLC project. In update mode, the framework can either regenerate the main logic by reusing the previously stored JSON mapping file or ask the user to redefine the bindings when required. This behaviour enables the controlled evolution of the main logic after updates in the process configuration or in the interfaces of the generated function blocks. As a result, the framework does not require the full manual reconstruction of the main program after each project modification.

Chapter 10

Appendix E

10.1 Update and consistency management

The proposed framework is designed not only to generate PLC projects from scratch, but also to support the controlled update of existing projects. This capability is particularly important in industrial environments, where automation systems frequently evolve due to process modifications, new equipment integration, or changes in control logic. To support this scenario, the generation pipeline adopts a consistency-preserving update strategy based on configurable execution modes and deterministic regeneration procedures.

10.1.1 Execution modes

The framework supports two global execution modes that determine how the generation pipeline interacts with an existing PLC project.

- **Create mode** is used when a new PLC project is generated from scratch.

  All required components, including hardware configuration, tags, function blocks, and the main control logic, are created automatically.

- **Upsert mode** is used when the framework operates on an already existing project.

  In this case, the generation procedures ensure that all required components exist in the project. If an element is missing it is created, while existing elements can be updated or replaced depending on the generation policy.

10.1.2 Block update strategy

At block level, the generation procedures adopt two possible policies:

- `CreateOnly`, where the generation fails if the block already exists.
- `Upsert`, where an existing block can be replaced by a newly generated version.

This mechanism allows the framework to regenerate parts of the PLC program without requiring a full reconstruction of the project.

10.1.3 Compilation-based validation

After the generation steps are completed, the framework performs an automatic save and compilation procedure. The PLC program is compiled using the compilation services provided by the TIA Portal Openness API. The compilation results are analysed in order to detect errors or inconsistencies in the generated project. Compilation messages are parsed and classified, allowing the framework to identify whether the generation process produced a valid PLC program. If compilation errors are detected, the generation process can be interrupted and the user is notified of the detected issues.

10.1.4 Consistency preservation during updates

When the framework is executed in update mode, several mechanisms are applied to preserve the structural consistency of the PLC project:

- regeneration of function blocks automatically recreates the associated instance data blocks.
- the main logic is regenerated using the stored parameter mapping configuration.
- tag creation procedures avoid duplicate logical addresses and tag names.
- naming utilities ensure that newly generated blocks receive unique identifiers.

These mechanisms allow the framework to update the PLC project while maintaining a coherent software architecture.

Bibliography

[1] Aintzane Armentia, Elisabet Estevez, Dario Orive, and Marga Marcos. A tool suite for automatic generation of modular machine automation projects. In *IEEE International Conference on Industrial Informatics (INDIN)*, pages 553–558. IEEE, 2018.

[2] Xiaoye Cai, Zhijian Jin, Hanyu Li, Alexander Kümpel, and Dirk Müller. Automated plc code generation for the implementation of mode-based control algorithms in buildings. *Buildings (Basel)*, 14(1):73, 2024.

[3] Joseph S. Hellewell, David Sanderson, Zi Wang, and Svetan Ratchev. Automated plc code generation in the connected morphing factory. In *IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1485–1490. IEEE, 2025.

[4] Angelica_Marcone. Original_author_work_created_for_this_dissertation. Unpublished.

[5] European Parliament. Industry 4.0. Technical report, European Parliament, 2016.

[6] Gopinath Karmakar. *Development of Safety-Critical Systems : Architecture and Software*. 1st ed. 2023.. edition, 2023.

[7] An Truong-Duc-Duy and Phuoc Vo Tan. A python framework for model-based design, commission and monitor the thermal process. In *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, 2021.

[8] Siemens AG. *SIMATIC STEP 7: Configuring Hardware and Communication Connections*. Siemens AG, 2017. TIA Portal.

[9] Diego Armando Giral-Ramírez, Andres Felipe Barrera-Cuestas, Marlón Mantilla-Castañeda, and Oscar Danilo Montoya-Giraldo. Temperature control using the simulink plc coder and the iec 61131 standard. *Scientia et technica*, 26(4):417–424, 2021.

[10] J. Zaytoon and B. Riera. Synthesis and implementation of logic controllers – a review. *Annual reviews in control*, 43:152–168, 2017.

[11] Siemens AG. *TIA Portal Openness: Automation of engineering workflows*. Siemens AG, 2023. TIA Portal.

[12] Factory I/O. Sample project for siemens s7-1200/1500, 2024.