FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Conversion of IEC 61131-3 LD Programs into SFC

Duarte Filipe de Fernandes e Tavares



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Mário Jorge Rodrigues de Sousa

June 20, 2026

© Duarte Tavares, 2026

Resumo

Os Controladores Lógicos Programáveis (PLCs) são amplamente utilizados na automação industrial, sendo as respetivas linguagens de programação normalizadas pela norma IEC 61131-3. O Ladder Diagram (LD) continua a ser uma das linguagens mais utilizadas devido à sua semelhança com os circuitos elétricos baseados em relés. Contudo, quando o comportamento sequencial é implementado através de realimentações e variáveis de estado, os programas em LD podem tornar-se difíceis de interpretar, validar e manter. Em contraste, o Sequential Function Chart (SFC) representa explicitamente este comportamento através de etapas, transições, ações e estruturas de ramificação.

Esta dissertação propõe e implementa uma abordagem de conversão automática de LD para SFC baseada no reconhecimento estrutural de padrões. O programa em LD é inicialmente traduzido para Structured Text (ST) através do Beremiz. Em seguida, uma versão estendida do compilador MATIEC constrói uma Abstract Syntax Tree (AST), cujas subárvores são comparadas com um catálogo de padrões representativos de estruturas sequenciais comuns. As correspondências identificadas geram fragmentos de SFC, posteriormente combinados através da agregação do SFC e da associação de ações. O SFC resultante é exportado em formato PLCopen XML.

O protótipo foi avaliado através de quatro testes de prova de conceito e de um caso de estudo retirado da literatura. Os programas XML gerados foram importados com sucesso no CODESYS. Os resultados demonstram que o reconhecimento de padrões baseado em ASTs permite identificar as estruturas de design e gerar os elementos de SFC correspondentes. Contudo, o caso de estudo também demonstra que o reconhecimento local de padrões, por si só, não permite reconstruir integralmente comportamentos dependentes de interações entre atribuições a diferentes variáveis. O desenvolvimento futuro requer, portanto, uma análise semântica parcial e/ou global e a validação formal do resultado gerado.

Palavras-chave: Controladores Lógicos Programáveis, Ladder Diagram, Sequential Function Chart, Abstratct Syntax Tree, reconhecimento de padrões, PLCopen XML.

Abstract

Programmable Logic Controllers (PLCs) are widely used in industrial automation, and their programming languages are standardized by IEC 61131-3. Ladder Diagram (LD) remains one of the most commonly used languages due to its similarity to relay-based electrical circuits. However, when sequential behavior is implemented through feedback paths and state variables, LD programs can become difficult to interpret, validate, and maintain. The Sequential Function Chart (SFC), in contrast, represents this behavior explicitly through steps, transitions, actions, and branching structures.

This dissertation proposes and implements an automatic LD-to-SFC conversion approach based on structural pattern matching. The LD program is first translated into Structured Text (ST) using Beremiz. An extended version of the MATIEC compiler then constructs an Abstract Syntax Tree (AST), whose subtrees are compared with a catalog of patterns representing common sequential structures. Successful matches generate SFC fragments that are subsequently combined through SFC aggregation and action association. The resulting SFC is exported in PLCopen XML format.

The prototype was evaluated through four proof-of-concept tests and a case study obtained from the literature. The generated XML programs were successfully imported into CODESYS. The results demonstrate that AST-based pattern matching can identify supported design structures and generate the corresponding SFC elements. However, the case study also shows that local pattern recognition alone cannot fully reconstruct the behavior that depends on interactions between assignments to different variables. Further development therefore requires parcial and/or global semantic analysis and formal validation of the generated output.

Key words: Programmable Logic Controllers, Ladder Diagram, Sequential Function Chart, Abstratct Syntax Tree, design patterns recognition, PLCopen XML.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Objective . . . 2
- 1.3 Dissertation Structure . . . 2

2 State of the Art **4**
- 2.1 Industrial PLC Programming and IEC 61131-3 . . . 4
  - 2.1.1 Evolution from Relay Logic to PLCs . . . 4
  - 2.1.2 IEC 61131 Architecture: Configurations, Resources, Tasks, and POUs . . . 5
  - 2.1.3 Programming Languages and Execution Model . . . 6
- 2.2 Program Representation in IEC 61131-3 . . . 7
  - 2.2.1 Variables, Data Types, and Memory Model . . . 7
  - 2.2.2 Program Organization Units: Functions, Function Blocks, and Programs . . . 9
- 2.3 Ladder Diagram . . . 11
  - 2.3.1 LD Structure: Rails, Rungs, Contacts, Coils, and Branches . . . 12
  - 2.3.2 LD Execution Semantics . . . 14
  - 2.3.3 Translation of LD Elements into Boolean Expressions . . . 15
  - 2.3.4 LD Elements with Memory . . . 15
  - 2.3.5 Function Blocks in LD . . . 17
  - 2.3.6 Structured Text as an Intermediate Representation for LD . . . 17
  - 2.3.7 Limitations of LD for Sequential Behavior . . . 19
- 2.4 Sequential Function Chart . . . 19
  - 2.4.1 Steps, Transitions, and Actions . . . 20
  - 2.4.2 SFC Execution Semantics . . . 21
  - 2.4.3 Action Qualifiers . . . 21
  - 2.4.4 Textual Representation of SFC Elements . . . 22
  - 2.4.5 Advantages of SFC for Sequential Behavior . . . 24
- 2.5 IEC 61131-3 Development Tools . . . 24
  - 2.5.1 PLCopen . . . 24
  - 2.5.2 XML-Based Representation and PLCopen XML . . . 25
  - 2.5.3 Beremiz and CODESYS . . . 29
  - 2.5.4 MATIEC Compiler Architecture . . . 29
- 2.6 Existing LD-to-SFC Conversion Approaches . . . 37
  - 2.6.1 Graph-Based and Design Recovery Approaches . . . 37
  - 2.6.2 Temporal-Logic and Plant-Model Approaches . . . 37
  - 2.6.3 State-Space Approaches . . . 38
  - 2.6.4 Lopes' Optimization of State-Space Approaches . . . 38
  - 2.6.5 Comparative Analysis . . . 39

3 Methodology **40**
3.1 Problem Definition and Context . . . 41
3.2 LD Representation through Structured Text . . . 42
3.3 AST Representation of the Input Program . . . 43
3.4 Pattern Definition and AST-Based Pattern Matching . . . 44
3.5 Post-Processing and PLCopen XML Export . . . 50

4 Development of the LD-to-SFC Conversion Tool **54**
4.1 Tool Requirements and Design Decisions . . . 54
4.2 Tool Architecture . . . 55
4.3 `generate_sfc` Module . . . 57
4.3.1 Module Organisation and Execution Flow . . . 57
4.3.2 Pattern Representation and Annotation . . . 65
4.3.3 AST Matching and Fragment Resolution . . . 65
4.3.4 Output Pattern and SFC Fragment Organization . . . 70
4.4 `generate_sfc_xml` Module . . . 72
4.4.1 Module Organisation and Execution Flow . . . 73
4.4.2 SFC AST Analysis and XML Planning . . . 79
4.4.3 PLCopen XML Serialization . . . 82

5 Test and Validation **83**
5.1 Evaluation Method . . . 83
5.2 Test Cases and Results . . . 84
5.2.1 Test 1: Single State and Associated Output . . . 84
5.2.2 Test 2: Sequential State Chain . . . 85
5.2.3 Test 3: Simultaneous Divergence . . . 87
5.2.4 Test 4: Selection Divergence . . . 88
5.3 Literature Case Study . . . 91
5.3.1 Original LD Program . . . 91
5.3.2 Extension of the Pattern Catalog . . . 93
5.3.3 Generated SFC . . . 99
5.4 Discussion of Results . . . 101

6 Conclusion **104**
6.1 Future Work . . . 105
6.2 Research Article . . . 106

References **107**

A Conversion Pattern Catalogue **109**

List of Figures

2.1 Software architecture defined by IEC 61131-3, showing Configurations, Resources, Tasks, and Program Organization Units. Taken from [12]. 6
2.2 Example of IEC 61131-3 Program Organization Units (Program, Function Block, and Function) and their respective structure declaration. Taken from [9]. 10
2.3 Equivalent representation of a simple FUNCTION in FBD (left) and ST (right). . 11
2.4 Equivalent representations of simple combinational control logic using different electrical schematics (left and center), and Ladder Diagram (right). Author's own illustration based on a simple instructional example. 12
2.5 Example Ladder Diagram illustrating IEC 61131-3 contact and coil symbols with reference numbering. 12
2.6 Example of memory-dependent rungs in Ladder Diagram using feedback contacts. 16
2.7 Example LD program containing a feedback-based internal state and a dependent output. 18
2.8 Example Sequential Function Chart illustrating steps, transitions, and associated actions. 20
2.9 MATIEC High-Level Program Architecture and Data Flow. 30

3.1 Overview of the proposed LD-to-SFC conversion pipeline. Reproduced from [14]. 41
3.2 Common LD patterns used to encode a state with activation and deactivation conditions. 43
3.3 AST representation of the ST assignment generated from an LD self-holding pattern. 44
3.4 Simplified AST representation of the top-level output pattern. The SFC network is initialized with `Step0`, while the `PATTERN_N__` node identifies the subpatterns whose generated SFC fragments may be inserted into the network. 46
3.5 LD state-encoding structure represented by pattern `p2`. 46
3.6 SFC fragment generated directly from subpattern `p2`. 48
3.7 SFC fragment generated indirectly from subpattern `p3`. 49
3.8 Internal processing stages of the LD-to-SFC conversion module. 50
3.9 SFC generated from the state assignments, including the simultaneous divergence from the step associated with `M1`. 52

4.1 Program flow of the implemented Stage 4 LD-to-SFC conversion modules. . . . 56
4.2 UML class diagram of the `generate_sfc.cc` module. 59
4.3 Sequence diagram of the `generate_sfc.cc` execution flow. 64
4.4 UML class diagram of the `generate_sfc_xml.cc` module. 74
4.5 Sequence diagram of the `generate_sfc_xml.cc` execution flow. 78

5.1 Input LD program for Test 1. 84
5.2 Generated SFC for Test 1 after import into CODESYS. 85

5.3 Input LD program for Test 2. 85
5.4 Generated SFC for Test 2 after import into CODESYS. 86
5.5 Input LD program for Test 3. 87
5.6 Generated SFC for Test 3 after import into CODESYS. 88
5.7 Input LD program for Test 4. 89
5.8 Generated SFC for Test 4 after import into CODESYS. 90
5.9 Neutralization-tank process used in the literature case study. Taken from [3] . . . 92
5.10 Original LD program of the neutralization-tank case study. Reproduced from [9] 92
5.11 SFC generated for the retained-state patterns detected in the case-study program. 99

List of Tables

2.1 Elementary Data Types Defined by IEC 61131-3 8
2.2 Variable Types Defined by IEC 61131-3 9
2.3 Ladder Diagram Coil Elements (numbered as in Fig. 2.5) 13
2.4 Ladder Diagram Contact Elements (numbered as in Fig. 2.5) 14
2.5 Sequential Function Chart structural elements 21
2.6 SFC action qualifiers defined by IEC 61131-3 22
2.7 Node forms with a predefined number of child references used by the MATIEC AST. 33

3.1 State dependencies extracted from the generated ST program. 52

4.1 Main visitors of the `generate_sfc` module and their role in the conversion pipeline. 60
4.2 Secondary visitors and helper structures of the `generate_sfc` module and the main visitors they support. 61
4.3 Main classes of the `generate_sfc_xml` module. 75

5.1 Mapping between case-study state variables and the added patterns. 98

List of Acronyms

AST	Abstract Syntax Tree
CODESYS	Controller Development System
EN	Enable
ENO	Enable Output
FB	Function Block
FBD	Function Block Diagram
IEC	International Electrotechnical Commission
I/O	Input/Output
ID	Identifier
IL	Instruction List
LD	Ladder Diagram
MATIEC	IEC 61131-3 compiler used as the basis for the implemented tool
PLC	Programmable Logic Controller
PLCopen	Organization defining open standards for industrial control software
POU	Program Organization Unit
RLL	Relay Ladder Logic
SFC	Sequential Function Chart
ST	Structured Text
UML	Unified Modeling Language
XML	eXtensible Markup Language
XSD	XML Schema Definition

Chapter 1

Introduction

Over the years, industrial automation started to heavily rely on programmable logic controllers (PLCs) to control machines, manufacturing lines and other process systems. These controllers are programmed using domain-specific languages standardized under the IEC 61131-3 standard, which defines both textual (Structured Text, Instruction List) and graphical (Ladder Diagram, Function Block Diagram, Sequential Function Chart) languages. Among these, the Ladder Diagram (LD) has historically been the most widespread, as it still is the most common graphical language in today's industry. Its symbolic structure closely resembles electrical relay circuits, which made it intuitive for engineers and electricians to execute on site. However, as systems grow in complexity, LD becomes less suitable for expressing sequential and state-based control behaviors as it shows a lot of limitations.

Sequential Function Charts (SFCs), also standardized by IEC 61131-3, provides a more structured, modular and readable approach to sequential control as it decomposes process logic into *steps* and *transitions*, clarifying the evolution of the system's state and making SFC better suited for complex industrial processes, where understanding and modifying control logic efficiently is critical. However, because LD has been extensively used for decades prior to the introduction of SFC, most of the industrial equipment still operates with LD programs, creating a substantial base of legacy software that is costly to adapt, maintain and modernize.

In this context, the present work focuses on LD programs that encode sequential or state-based control behavior that can be represented through SFC steps, transitions, and actions. The proposed methodology is therefore not intended as a general-purpose conversion method applicable to every LD program. Its scope is restricted to programs whose relevant control structures can be identified through the supported patterns and mapped to corresponding SFC structures. This distinction is essential, as the use of LD alone does not imply that a program contains a sequence of operating states suitable for reconstruction as an SFC.

1.1 Motivation

The motivation for this research arises from a technological gap: the absence of automatic and reliable tools capable of translating existing LD programs into their equivalent SFC representations. Existing methods are often incomplete, dependent on manual intervention, or computationally heavy. To address this, the present work proposes a conversion methodology that bridges the legacy language of LD-based control systems with the structured modeling advantages of SFC. This contributes not only to the modernization of industrial automation codebases but also to improved maintainability, readability, and integration with the diverse open platforms currently available in the market.

1.2 Objective

The objective of this dissertation is to design and implement a conversion methodology that automatically translates limited LD programs into their equivalent SFC representations. The proposed approach is based on identifying common design patterns used to represent sequential or state-based behavior in LD programs and detecting them through pattern matching algorithms applied to Abstract Syntax Trees (ASTs) derived from each program analysis. The expected outcome is a validated proof-of-concept tool capable of parsing, recognizing, and mapping LD structures into standardized SFC elements, thereby facilitating the transition towards modern, modular and transparent industrial control software.

1.3 Dissertation Structure

The remainder of this dissertation is organized as follows.

Chapter 2 presents the state of the art relevant to the automatic conversion of Ladder Diagram (LD) programs into Sequential Function Charts (SFC). Introduces the IEC 6131-3 programming model, the main characteristics and execution semantics of LD, Structured Text (ST) and SFC, and the use of Abstract Syntax Trees (ASTs) as program representations. Existing LD-to-SFC conversion methodologies are reviewed and compared.

Chapter 3 describes the proposed conversion methodology. Explains the use of ST as an intermediate representation, the organization of the conversion patterns, and the established correspondence between LD constructs, ST AST subtrees, and SFC fragments. The chapter presents the main transformation stages, including pattern detection, state-variable identification, fragment generation, dependency handling, and construction of the resulting SFC AST.

Chapter 4 presents the implementation of the proposed methodology as an extension of the MATIEC compiler infrastructure. It describes the relevant internal AST structures, the pattern-matching and transformation mechanisms implemented in `generate_sfc.cc`, and the generation of the final PLCopen XML representation through `generate_sfc_xml.cc`.

Chapter 5 evaluates the conversion process developed through proof-of-concept examples and a case study taken from the LD-to-SFC literature. The chapter analyzes the generated ST, SFC AST and PLCopen XML output, discusses the correspondence between the detected LD structures and the resulting SFC fragments, and identifies the current capabilities and limitations of the methodology.

Chapter 6 presents the final considerations of the dissertation. It summarizes the main contributions and results, discusses the limitations of the current implementation, and outlines possible directions for future work.

The appendices provide the UML class diagrams of the developed conversion modules and the complete conversion-pattern catalog used during the evaluation.

Chapter 2

State of the Art

2.1 Industrial PLC Programming and IEC 61131-3

The current edition of IEC 61131-3 was published in 2025. However, unless otherwise stated, this chapter follows the third edition, published in 2013, as this was the edition considered during the development of the tool and in most of the analyzed literature [5, 6].

2.1.1 Evolution from Relay Logic to PLCs

Before the introduction of international standards for industrial automation, Programmable Logic Controllers (PLCs) were programmed using proprietary languages and tools developed independently by each manufacturer. These systems were tightly attached to specific hardware architectures and required dedicated training for engineers and technicians, resulting in limited interoperability between platforms. As a consequence, migrating control applications between vendors or upgrading existing installations often required significant retraining efforts, increased engineering time, and high maintenance costs, ultimately leading to inefficiencies in both development and operation [7].

The IEC 61131 standard was developed to address these limitations by defining a common framework for PLC systems, including hardware and software requirements necessary to ensure a minimum level of compatibility across compliant platforms.

The standard is intended to be a flexible guideline for PLC programming rather than a strict prescriptive standard. Consequently, the standard defines a set of features that manufacturers are required to declare as supported or unsupported by their implementations. This flexibility promotes the unification of PLC designs while preserving a shared conceptual and architectural foundation among different manufacturers.

By unifying programming models and design principles, IEC 61131 enables the convergence of development practices across the industrial automation market. Manufacturers benefit from reduced development costs through shared tooling and reusable components, while software developers can exchange and reuse application modules across projects. From the end-user perspective,

the standard lowers training requirements and facilitates system evolution by easing the integration or replacement of components from different manufacturers.

The standard is structured into multiple parts, each part addressing a specific aspect of PLC systems. IEC 61131-1 establishes basic terminology and core concepts. IEC 61131-2 specifies hardware requirements, including electronic and mechanical characteristics, together with the corresponding qualification tests. IEC 61131-3, which is reviewed in detail throughout this chapter, focuses on the software aspects of PLC systems, defining the internal software architecture and the set of standardized programming languages. IEC 61131-4 provides guidance for PLC users regarding system selection, installation, and maintenance. IEC 61131-5 details communication aspects, defining messaging services based on the Manufacturing Automation Protocol (MAP). IEC 61131-6 covers communication using fieldbus technologies. IEC 61131-7 introduces software elements to support fuzzy logic within PLC applications, and IEC 61131-8 presents implementation guidelines for the programming languages defined in IEC 61131-3.

2.1.2 IEC 61131 Architecture: Configurations, Resources, Tasks, and POUs

IEC 61131-3 introduces a modular software architecture that enhances the portability, reusability, and maintainability of industrial control applications. It defines a hierarchy of Program Organization Units (POUs): Programs, Function Blocks (FBs), and Functions, enabling the decomposition of applications into well-defined and reusable components. This architecture is supported by strong data typing, structured data representations, and five standardized programming languages for implementation.

At the top level, a configuration contains one or more resources, each executing one or more tasks that schedule programs. Programs are made up of variables, function blocks and functions, which encapsulate logic and data to promote modular design. Variables can be local, global or directly mapped to I/O channels, and data exchange occurs through variable control paths and access paths, as shown in Figure 2.1.

This layered architecture establishes a clear separation between execution control and data flow, supporting scalable system design and facilitating advanced analysis tasks such as design pattern matching across POUs [12].

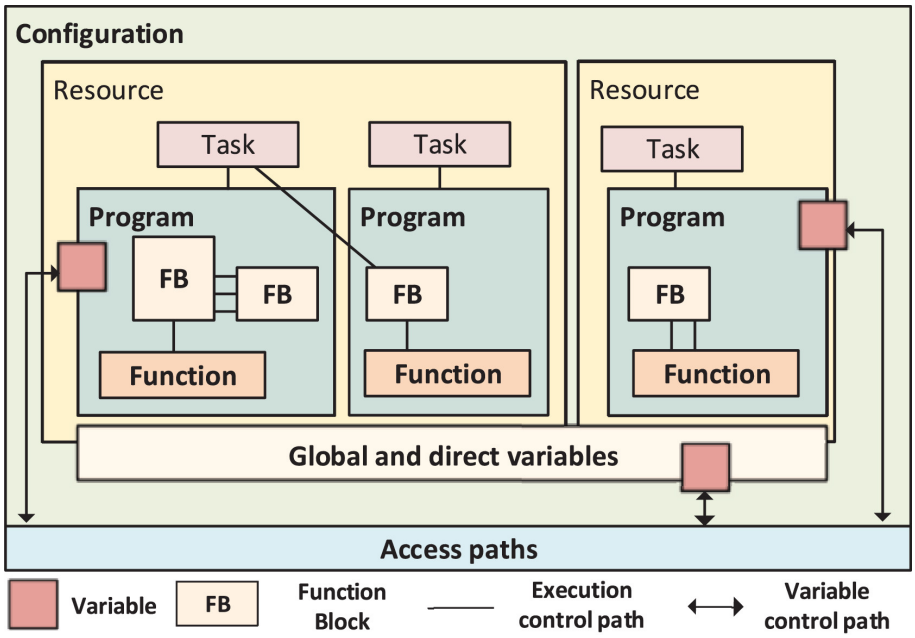


Figure 2.1: Software architecture defined by IEC 61131-3, showing Configurations, Resources, Tasks, and Program Organization Units. Taken from [12].

2.1.3 Programming Languages and Execution Model

From the set of five defined languages, IEC 61131-3 defines two textual languages, Instruction List (IL) and Structured Text (ST), and three graphical languages, Function Block Diagram (FBD), Ladder Diagram (LD), and Sequential Function Chart (SFC).

IL is a low-level, assembly-like language characterized by a simple and compact instruction set. In contrast, ST is a high-level language, syntactically similar to Pascal, which enables the development of more complex and elaborate control logic. LD is a graphical language derived from relay-based schematics, in which Boolean logic is represented through contacts and coils. FBD is a graphical language that expresses control logic through the interconnection of function blocks. SFC is a graphical language intended for structuring control applications into discrete steps, each associated with specific actions, with transitions defining the conditions under which execution progresses between steps.

A PLC application is normally executed cyclically: inputs are read, the configured tasks execute the associated programs, internal variables and outputs are updated, and the cycle repeats. This scan-cycle execution model is particularly relevant for LD programs, since feedback contacts, set/reset coils, edge-sensitive elements, and function block instances may depend on values computed in previous cycles rather than only on the current input parameters [8].

2.2 Program Representation in IEC 61131-3

2.2.1 Variables, Data Types, and Memory Model

IEC 61131-3 defines a complete variable declaration model for information storage, specifying both the data type and the variable type associated with each variable. The data type determines the kind of information that can be stored, while the variable type defines how the variable can be accessed.

The standard defines a wide range of elementary data types, including Boolean, bitstring, integer, real, time and date, and string types. In addition to these elementary types, IEC 61131-3 supports the definition of derived data types, allowing developers to create new types based on existing ones. This feature promotes stronger typing, improved readability, and better structuring of control applications.

Generic data types are also defined. These types, identified by the prefix `ANY_`, represent abstract categories of data types, particularly useful for writing reusable Functions and Function Blocks that are independent of a specific elementary data type.

Table 2.1: Elementary Data Types Defined by IEC 61131-3

Keyword	**Description**
Boolean and Bitstring Types	
BOOL	Boolean value
BYTE	8-bit bitstring
WORD	16-bit bitstring
DWORD	32-bit bitstring
LWORD	64-bit bitstring
Integer Types	
SINT	Signed short integer (8-bit)
USINT	Unsigned short integer (8-bit)
INT	Signed integer (16-bit)
UINT	Unsigned integer (16-bit)
DINT	Signed double integer (32-bit)
UDINT	Unsigned double integer (32-bit)
LINT	Signed long integer (64-bit)
ULINT	Unsigned long integer (64-bit)
Real Types	
REAL	Single-precision floating-point value (32-bit)
LREAL	Double-precision floating-point value (64-bit)
Time and Date Types	
TIME	Time interval value
DATE	Date value
TIME_OF_DAY (TOD)	Time of day
DATE_AND_TIME (DT)	Date and time value
Character and String Types	
CHAR	Single-byte character
WCHAR	Double-byte character
STRING	Variable-length single-byte character string
WSTRING	Variable-length double-byte character string

The PLC memory is divided into three regions: input memory, internal memory, and output memory. Input memory holds data acquired from analog and digital input modules, while output memory contains data destined for output channels. Internal memory is used to store intermediate data generated during program execution. Access to these memory regions can be performed directly or through variable mappings [8].

The variable type also determines how a variable is accessed within a PLC application, defining whether it is used as an input parameter, output parameter, or both. Variables may also differ in scope, being local to a POU, internal to its execution context, or globally accessible within the whole program.

Table 2.2: Variable Types Defined by IEC 61131-3

Variable Type	Description
VAR	Defines a local variable accessible only within the POU in which it is declared.
VAR_IN	Defines an input variable of a POU. These variables are passed as input parameters when the POU is invoked and can only be read within the POU.
VAR_OUT	Defines an output variable of a POU. The declaring POU can read and write these variables, while other POUs can only read them.
VAR_IN_OUT	Combines input and output behavior. Both the calling POU and the called POU can read and write the variable, effectively implementing pass-by-reference semantics.
VAR_GLOBAL	Defines global variables that can be declared at the configuration, resource, or program level and accessed by POUs within the same scope.
VAR_EXTERNAL	Allows a POU to access a global variable declared elsewhere by referencing a variable with the same name and data type.
VAR_ACCESS	Defines access variables that create access paths to input/output variables, global variables, or directly represented variables, enabling access by remote devices.
VAR_TEMP	Defines temporary local variables whose memory is allocated for the duration of a single POU execution and cleared afterward.

The standard also specifies a set of attributes that can further constrain variables. The `RETAIN` attribute indicates that a variable must be stored in non-volatile memory, ensuring that its value is preserved across power cycles or loss of power. The `CONSTANT` attribute specifies that a variable is immutable after initialization, preventing subsequent modification. The `AT` attribute allows a variable to be stored within a specific memory location or physical input/output address.

2.2.2 Program Organization Units: Functions, Function Blocks, and Programs

Program Organization Units (POUs) are the fundamental software building blocks defined by IEC 61131-3 for structuring control applications. They are the smallest independent units in an IEC 61131-3 program, encapsulating both data and behavior, and can be executed independently of other POUs.

IEC 61131-3 defines three types of POUs: *Functions*, *Function Blocks*, and *Programs*. These units provide the structural foundation for modular and hierarchical software development. Each POU is defined by a header and a footer and consists of two main parts: a variable declaration section and a code body.

While all POUs share this structural organization, their behavior and execution semantics differ depending on the POU type.

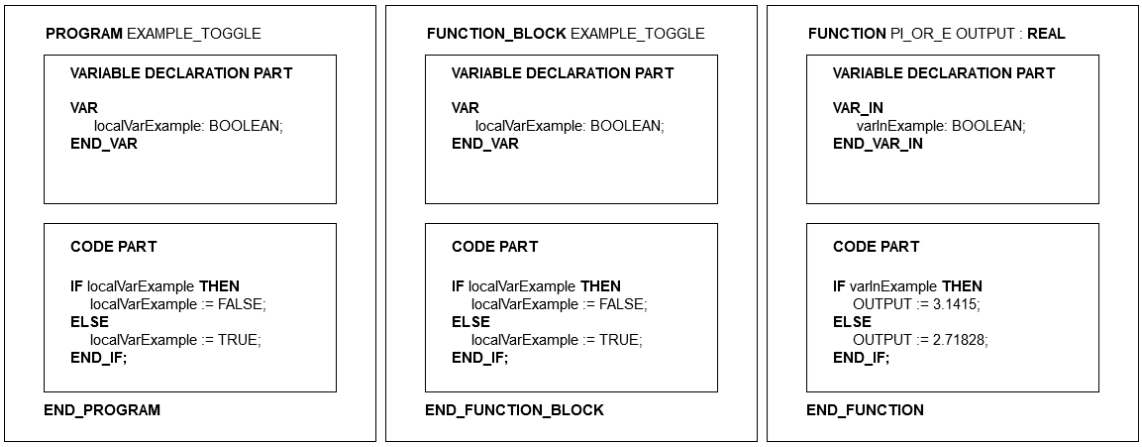


Figure 2.2: Example of IEC 61131-3 Program Organization Units (Program, Function Block, and Function) and their respective structure declaration. Taken from [9].

Functions represent the most elementary type of POU in IEC 61131-3. A Function POU accepts one or more input parameters, declared as `VAR_IN` or `VAR_IN_OUT`, and produces exactly one return value. This return value, together with its data type, is explicitly specified in the POU header.

Functions represent stateless computational units that produce a single output based solely on their input values. They do not retain internal memory between execution cycles and are typically used for arithmetic operations, logical expressions, and data transformations. As a result, functions exhibit deterministic behavior that is independent of previous execution states.

Regarding interaction with other POUs, a Function may only invoke other Functions and is prohibited from calling Function Blocks or Programs. Functions cannot be instantiated and are only executed when explicitly called. Once defined, they are globally accessible to other POUs within the application.

IEC 61131-3 also defines implicit control variables associated with Functions. The `EN` input enables or disables execution: when set to `FALSE`, the Function is not executed and the output value stays unassigned, whereas setting `EN` to `TRUE` triggers execution and updates the output value. The corresponding `ENO` output signals whether the Function was executed successfully. This mechanism supports the sequential use of Functions by propagating execution conditions from one Function to the next.



Figure 2.3: Equivalent representation of a simple FUNCTION in FBD (left) and ST (right).

While Functions are stateless and produce output solely as a function of their inputs, Function Blocks extend this concept by maintaining internal state through persistent variables. By retaining data between execution cycles, Function Blocks can represent memory-dependent behavior, including timers, counters, latches, and control algorithms. Consequently, they are suitable for modeling dynamic processes and control components whose behavior evolves over time. FBs can be defined using any of the languages available in the standard.

Similar to variables, Function Blocks (FBs) must be instantiated in the variable declaration section of a POU. Once instantiated, FBs may be invoked within other Function Blocks or within PROGRAMs. FB instances can be declared as global or accessed through `VAR_EXTERNAL`. The `RETAIN` attribute may be applied to an FB instance to preserve its internal state or variables across execution cycles. Multiple instances of the same FB type can be created within a FB or a PROGRAM, each operating independently. Interaction with a Function Block occurs exclusively through the variables defined in its interface. Each FB is encapsulated and operates independently of external events, which leads to comparisons with object-oriented programming paradigms, even though mechanisms such as inheritance and polymorphism are not supported.

PROGRAMs represent the highest level of POUs and define the executable control logic associated with a task. A PROGRAM typically coordinates the execution of multiple Function Blocks and Functions, interacting with global variables and mapped input/output signals. Although PROGRAMs may also declare internal variables, their primary role is to organize and orchestrate lower-level POUs within the execution context defined by tasks and resources.

2.3 Ladder Diagram

Of the five languages defined by IEC 61131-3, Ladder Diagram (LD) became one of the most widespread in industry due to its close resemblance to traditional electrical wiring diagrams. Its intuitive structure allowed electricians and engineers to adopt PLC programming without extensive retraining. However, the simplicity that made LD accessible also introduced significant limitations when the language is used to represent sequential behaviour. LD is naturally suited to boolean and combinational logic, while sequencing, state retention, and concurrency are usually encoded indirectly through variable dependencies, feedback paths, and memory elements. As industrial processes grew in complexity, LD programs became harder to interpret, verify, maintain, and test [7].

At the structural level, a Ladder Diagram program is composed of a set of rungs arranged between two vertical power rails. Each rung represents a logic network whose evaluation determines

the activation of one or more output variables. Figure 2.4 illustrates the relationship between traditional electrical schematics and their equivalent LD representation.

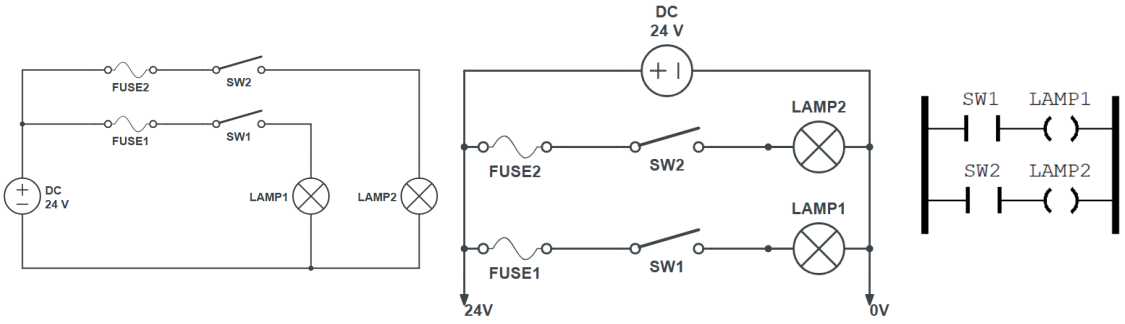


Figure 2.4: Equivalent representations of simple combinational control logic using different electrical schematics (left and center), and Ladder Diagram (right). Author's own illustration based on a simple instructional example.

2.3.1 LD Structure: Rails, Rungs, Contacts, Coils, and Branches

A Ladder Diagram is organized as a set of networks, usually called rungs, connected between the left and right power rails. Each rung is evaluated from left to right, according to the order in which contacts, branches, function calls, and output elements are allocated. The resulting rung arrangement determines whether the associated coil or output operation is executed.

The basic graphical elements of LD consist of contacts and coils. Contacts represent Boolean conditions associated with input, output, or internal variables, and may be normally open, normally closed, or edge-sensitive. Coils represent operations applied to Boolean variables and may implement different behaviors, such as direct assignment, negated assignment, set/reset actions, or transition-sensitive operations.

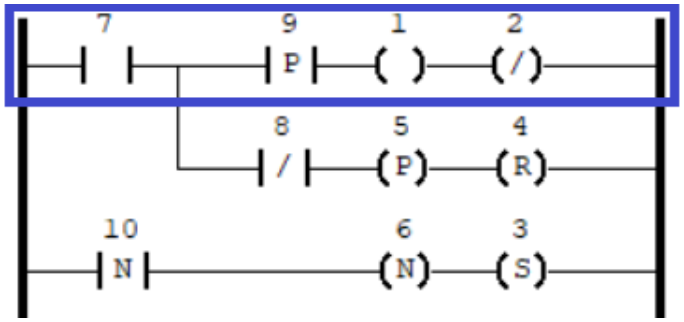


Figure 2.5: Example Ladder Diagram illustrating IEC 61131-3 contact and coil symbols with reference numbering.

Although IEC 61131-3 defines the semantics of these elements, their graphical representation is largely standardized in practice by IEC-compliant development environments. In this work, LD symbols are illustrated using the notation adopted by the Beremiz automation environment, which provides an open-source implementation aligned with the IEC 61131-3 specification. The coil

and contact symbols illustrated in Tables 2.3 and 2.4 follow the notation adopted by IEC 61131-3 compliant environments and are illustrated using the Beremiz toolchain [1].

Table 2.3: Ladder Diagram Coil Elements (numbered as in Fig. 2.5)

#	Symbol	Coil type	Description
1	–( )–	Coil	The Boolean value arriving from the left side of the rung is assigned to the associated variable.
2	–(/)–	Negated coil	The inverse of the Boolean value arriving from the left side of the rung is assigned to the associated variable.
3	–(S)–	Set coil	The associated variable is set to `TRUE` when the rung evaluates to `TRUE` and remains set until explicitly reset.
4	–(R)–	Reset coil	The associated variable is set to `FALSE` when the rung evaluates to `TRUE` and remains reset until explicitly set.
5	–(P)–	Positive transition-sensing coil	The associated variable is set to `TRUE` for a single execution cycle when a rising transition is detected.
6	–(N)–	Negative transition-sensing coil	The associated variable is set to `TRUE` for a single execution cycle when a falling transition is detected.

Table 2.4: Ladder Diagram Contact Elements (numbered as in Fig. 2.5)

#	Symbol	Contact type	Description
7		Normally open contact	The logical state propagates from left to right when the associated Boolean variable evaluates to `TRUE`.
8		Normally closed contact	The logical state propagates from left to right when the associated Boolean variable evaluates to `FALSE`.
9		Positive transition-sensing contact	The logical state is propagated for a single evaluation cycle when the variable transitions from `FALSE` to `TRUE`.
10		Negative transition-sensing contact	The logical state is propagated for a single evaluation cycle when the variable transitions from `TRUE` to `FALSE`.

The boolean condition associated with a rung is obtained from its graphical branching structure. Contacts placed sequentially on the same path are combined using `AND`, while alternative branches that rejoin before the same output operation are combined using `OR`. Thus, two normally open contacts on the same path represent $A \wedge B$, whereas two normally open contacts on alternative branches represent $A \vee B$. Normally closed contacts contribute the negated value of their associated variable. This interpretation is useful for extracting the rung enabling condition, but it does not fully define the rung behavior, since the operation performed by the output element must also be considered, according to the coil type defined.

2.3.2 LD Execution Semantics

In the absence of elements that alter control flow, the evaluation of a Ladder Diagram program proceeds sequentially from top to bottom, rung by rung, with each rung evaluated from left to right.

The standard defines rules to ensure that execution remains unambiguous and consistent. In particular, all input values must be available before an element is evaluated; for example, all contacts in a rung must be evaluated before the corresponding coil is processed. The evaluation of an element is considered complete only once all its outputs have been determined, and the evaluation of the entire network is complete only after all output coils have been evaluated. When data is transferred between rungs, all values from the preceding rung must be available before the evaluation of the subsequent one begins.

Ladder Diagrams also define jumps and labels, which enable the transfer of control from one part of the program to another. However, the IEC 61131-3 guidelines discourage their use, as the

standard does not formally specify the evaluation behavior of rungs following a jump instruction when it is activated, even though it is generally assumed that those rungs are not evaluated. For this reason, jump and label constructs are not considered as primary targets in the conversion approach discussed in this work.

The scan-cycle execution model is particularly important for sequential behaviour. In LD, state is often not represented explicitly as a named state of the controller, but rather emerges from the values stored in Boolean variables, coils, memory elements, and function block instances. Therefore, two visually similar rung structures may have different meanings depending on whether their output is a direct coil, a negated coil, a set/reset coil, or an element with memory.

2.3.3 Translation of LD Elements into Boolean Expressions

The LD elements shown in the blue-highlighted part of Fig. 2.5 can be represented as a Boolean expression by first extracting the enabling condition of the contact network and then applying the corresponding coil operations.

In this example, the contact network contains a normally open contact, represented here as X_7, and a positive transition-sensing contact, represented as $\uparrow X_9$. The enabling condition of the highlighted branch is therefore:

$$X_7 \wedge \uparrow X_9$$

This condition is then applied to two different output elements. Coil 1 is a normal coil, so its associated variable receives the value of the condition. Coil 2 is a negated coil, so its associated variable receives the inverse value. In ST-like notation, the highlighted rung can therefore be expressed as:

$$Y_1 = X_7 \wedge \uparrow X_9;$$
$$Y_2 = \neg(X_7 \wedge \uparrow X_9);$$

This example illustrates the basic principle used in the conversion process: the contact network is translated into a boolean condition, and each output element is then translated according to its own semantics.

2.3.4 LD Elements with Memory

Complex control behavior in LD is achieved by combining basic elements on multiple rungs and often relying on feedback paths, self-holding coils, and interlocking contacts. These constructions do not explicitly represent control states, but instead encode them implicitly through boolean variable dependencies distributed across all the rungs that make up the program [3, 9, 15].

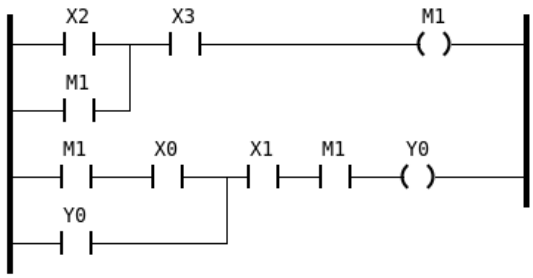


Figure 2.6: Example of memory-dependent rungs in Ladder Diagram using feedback contacts.

Figure 2.6 illustrates this type of memory-dependent behavior. In the first rung, variable $M1$ is activated when $X2$ and $X3$ are true. Once $M1$ becomes active, its own contact provides an alternative path that keeps the coil energized, even if the original activation condition $X2$ is no longer true. This is a normal feedback structure, where the current value of the variable depends on its value in the previous PLC scan cycle.

The behavior of the first rung can be represented abstractly as:

$$M1_{k+1} = X3 \wedge (X2 \vee M1_k),$$

where $M1_k$ is the value of $M1$ in the previous scan cycle and $M1_{k+1}$ is the value assigned during the current scan. This expression shows why feedback-based LD logic cannot be treated as purely combinational: the current value depends explicitly on a previously stored value.

The second rung follows the same principle. The output $Y0$ is enabled by $M1$, $X0$, and $X1$, but it also contains a feedback contact of $Y0$, allowing the output to remain active after the initial branch condition has been satisfied. Its behavior can be represented as:

$$Y0_{k+1} = M1 \wedge X1 \wedge (X0 \vee Y0_k).$$

In both cases, the feedback contact acts as an implicit memory element. The LD program does not declare an explicit internal variable or state, but the variables $M1$ and $Y0$ behave as stored states because their next values depend on their previous values.

Set/reset coils provide another direct form of memory. A set coil stores `TRUE` in the associated variable when its enabling condition is true and preserves that value until a reset operation is executed. In contrary, a reset coil stores `FALSE` and preserves that value until a set operation is executed. When set and reset conditions are implemented in different rungs, the final value may also depend on rung order, since the program is evaluated sequentially from top to bottom.

Edge-sensitive contacts and coils also require memory, because their behavior depends on the comparison between the current and previous values of a variable. A positive transition-sensing contact is true only when a variable changes from `FALSE` to `TRUE`; a negative transition-sensing contact is true only when a variable changes from `TRUE` to `FALSE`. Therefore, edge detection requires the previous value of the signal to be retained between scan cycles.

These memory-dependent elements are particularly relevant to this work because they are the LD constructs most closely related to explicit SFC states and transitions. Feedback paths, set/reset coils, latches, flip-flops, and edge elements often encode the same type of sequential behavior that SFC represents directly through active steps, guarded transitions, and actions [3, 9, 15].

2.3.5 Function Blocks in LD

In addition to contacts and coils, IEC 61131-3 also permits the use of Functions and Function Blocks within LD rungs, provided that the signals connected to the rung are of Boolean type. This allows LD programs to incorporate higher-level computational or control elements while preserving the Boolean execution model of the language.

Function Blocks are especially important because, unlike Functions, they may retain internal state between execution cycles. In LD programs, timers, counters, and SR/RS blocks are common examples of stateful Function Blocks. Timers retain elapsed time and update their output depending on both the input condition and the configured time parameters. Counters retain an accumulated count and update it according to rising or falling events. SR and RS blocks implement set/reset memory with a defined priority between set and reset inputs.

From a conversion perspective, these Function Blocks cannot be reduced to simple Boolean expressions without losing behaviour. Their outputs may depend on internal variables, previous scan values, elapsed time, or event history. Therefore, they must be treated as explicit memory-dependent constructs in any LD-to-SFC representation.

2.3.6 Structured Text as an Intermediate Representation for LD

Structured Text (ST) is one of the textual programming languages defined by IEC 61131-3. It is a high-level language whose syntax is close to Pascal-like programming languages, supporting assignments, Boolean and arithmetic expressions, conditional statements, iteration statements, Function calls, and Function Block invocations. Compared with graphical languages such as LD, FBD, and SFC, ST provides a compact textual notation that is well suited for describing complex control logic and related data manipulation.

The role of ST in IEC 61131-3 is not limited to manual programming. Because it is textual and structured, ST can also serve as a convenient representation for program analysis and transformation. Graphical languages express program logic through visual elements, connections, and coordinates, while ST expresses logic through statements and expressions. This makes ST easier to parse using conventional compiler techniques, since assignments, variables, operators, and Function Block calls are represented directly in the program text.

This property is relevant when analysing LD programs. A Ladder Diagram rung can often be represented as an ST statement by translating its contact network into a Boolean expression and then applying the corresponding output operation. Contacts placed sequentially contribute `AND` operations, alternative branches contribute `OR` operations, normally closed contacts contribute

negated variables, and coils are represented by assignments or memory-related operations. Therefore, ST can express the logical result of an LD rung without relying on the original graphical layout.

However, this translation must preserve the execution semantics of the LD program. LD programs are evaluated rung by rung, and the value assigned in one rung may be used in later rungs during the same scan cycle. For this reason, the order of the generated ST statements is significant. Similarly, feedback contacts must be represented in a way that preserves their memory-dependent behavior. A feedback-based rung is not simply a combinational Boolean equation; it represents an assignment where the next value of a variable depends on its previous value.

Fig. 2.7 illustrates a simple LD program containing a self-holding structure and a dependent output. The first rung defines the internal variable `M0` using an activation contact `X0`, a deactivation contact `X1`, and a feedback contact of `M0`. The second rung assigns output `Y0` according to the value of `M0`. An equivalent ST representation of this LD structure is shown in Listing 2.1.

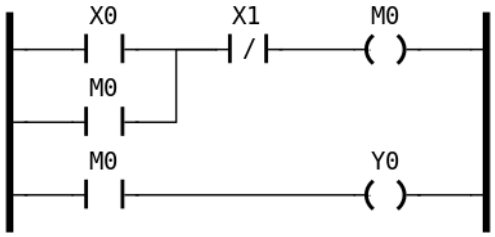


Figure 2.7: Example LD program containing a feedback-based internal state and a dependent output.

Listing 2.1: ST representation of the LD program shown in Fig. 2.7.

```
PROGRAM program0
  VAR_INPUT
    X0 : BOOL;
    X1 : BOOL;
  END_VAR

  VAR
    M0 : BOOL;
  END_VAR

  VAR_OUTPUT
    Y0 : BOOL;
  END_VAR

  M0 := NOT X1 AND (M0 OR X0);
  Y0 := M0;
```

```
END_PROGRAM
```

In Listing 2.1, the assignment to `M0` preserves the self-holding behaviour of the first LD rung. The variable `M0` appears both as the assignment output and inside the expression that determines its next value, exposing the feedback dependency in textual form. The following assignment, `Y0 := M0`, preserves the dependency between the internal memory variable and the output. This example shows how ST can represent LD-derived logic in a form that retains rung order, variable dependencies, and feedback structures while remaining suitable for compiler-based and automatic analysis.

2.3.7 Limitations of LD for Sequential Behavior

LD remains highly effective for representing Boolean and combinational control logic. Its graphical notation is simple, familiar to engineers and technicians with experience in relay-based systems, and well suited for small to medium-sized control tasks involving interlocks, permissive, and direct input-output relations.

However, when LD is used to implement sequential behavior, the resulting program may become difficult to understand. The state of the system is not represented as an explicit control state, but rather distributed across coils, auxiliary variables, feedback contacts, set/reset elements, timers, counters, and interlocking conditions. As a result, the sequence of operation is often mixed with the application logic, making the program harder to analyse, document, verify, and modify [3, 9, 15].

This limitation is the central motivation for converting LD programs into Sequential Function Chart representations. While LD encodes state implicitly through scan-cycle effects and memory-dependent constructs, SFC represents states and transitions explicitly. A systematic conversion methodology must therefore identify the LD constructs that encode stateful behaviour, translate their semantics into an intermediate representation, and reconstruct the corresponding SFC steps, transitions, and actions.

2.4 Sequential Function Chart

Sequential Function Chart (SFC) is the IEC 61131-3 language specifically intended for representing sequential control logic. Unlike LD, SFC represents the evolution of the control process explicitly through steps, transitions, and actions.

An SFC program can be understood as a directed control structure in which steps represent the possible active states of the controller and transitions define the Boolean conditions that allow execution to progress from one step to another. At any instant, one or more steps may be active, depending on the structure of the chart and on the execution semantics defined by the standard.

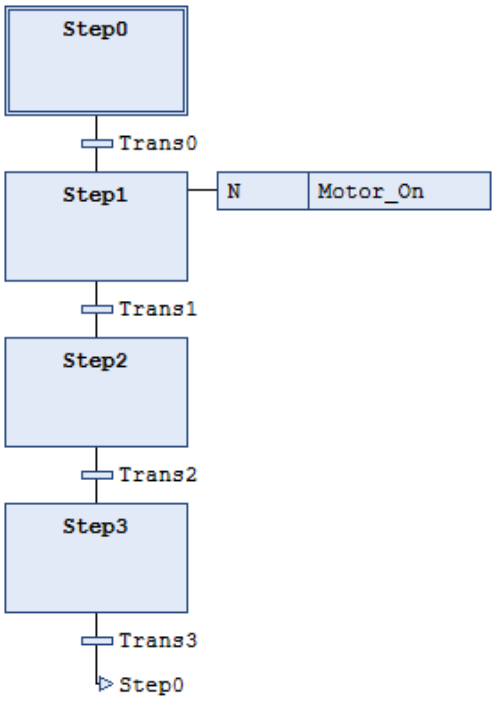


Figure 2.8: Example Sequential Function Chart illustrating steps, transitions, and associated actions.

Figure 2.8 presents a simple SFC composed of a sequence of steps connected by transition conditions. Each step remains active while its outgoing transition condition is false. When the transition condition evaluates to `TRUE`, the current step is deactivated and the following step is activated. This explicit representation of state progression contrasts with LD programs, where the same sequential behavior may be distributed across several rungs and encoded through Boolean dependencies between variables [3, 9].

2.4.1 Steps, Transitions, and Actions

The basic structural elements of an SFC are steps, transitions, and actions. A step represents a control state or phase of operation. Each SFC contains at least one initial step, which defines the starting point of execution. A transition connects one or more preceding steps to one or more following steps and is associated with a Boolean condition. Actions define the operations associated with steps and may be written in any of the other IEC 61131-3 languages.

Table 2.5: Sequential Function Chart structural elements

Element	Description
Step	Represents a discrete control state or phase. A step may be active or inactive and may have one or more associated actions.
Initial step	Defines the starting state of the SFC and is activated when the chart is initialized.
Transition	Defines a Boolean condition that controls the progression from preceding steps to following steps.
Action	Defines the operation associated with a step. Actions may be written in another IEC 61131-3 language and executed according to an action qualifier.
Divergence / Convergence	Defines alternative or simultaneous execution paths, allowing branching, parallelism, and synchronization.

This organization separates the state structure of the program from the operations executed in each state. Steps and transitions describe the evolution of the control process, while actions describe the behaviour associated with each active step.

2.4.2 SFC Execution Semantics

The execution of an SFC is based on the activation and deactivation of steps. A transition is enabled when all immediately preceding steps are active. If the transition is enabled and its Boolean condition evaluates to `TRUE`, the transition fires. As a result, the preceding steps are deactivated and the following steps are activated.

This execution model makes the state of the controller explicit. Instead of being inferred from internal Boolean variables, as often happens in LD, the active step or set of active steps directly represents the current phase of the sequential process. This characteristic is particularly relevant for LD-to-SFC conversion, since the objective is to recover the implicit state structure encoded in LD and represent it through explicit SFC steps and transitions.

SFC also supports alternative and simultaneous paths. Alternative paths allow different execution branches to be selected according to transition conditions, while simultaneous paths allow several branches to be active at the same time. These constructs make SFC suitable for representing not only linear sequences, but also processes involving branching, synchronization, and concurrent control phases.

2.4.3 Action Qualifiers

Actions associated with SFC steps may be controlled by action qualifiers. These qualifiers define how an action behaves relative to the activation of the associated step. In the absence of an explicit qualifier, the normal qualifier `N` is assumed, meaning that the action is executed while the step is active.

Table 2.6: SFC action qualifiers defined by IEC 61131-3

Qualifier	Description
N	Normal action. The action is executed while the associated step is active.
S	Stored action. The action starts when the associated step becomes active and remains active until it is explicitly reset.
R	Reset action. The action is reset when the associated step is active.
L	Time-limited action. The action is executed while the step is active, but only for a specified maximum duration.
D	Delayed action. The action starts only after a specified delay, provided that the associated step is still active.
P / P1	Pulse action on step activation. The action is executed once when the associated step becomes active.
P0	Pulse action on step deactivation. The action is executed once when the associated step becomes inactive.
SD	Stored and delayed action. The action starts after a specified delay and then remains active until reset.
DS	Delayed and stored action. The action is stored only if the step remains active for the specified delay.
SL	Stored and time-limited action. The action starts when the step becomes active and remains active for a specified duration, even if the step becomes inactive.

For LD-to-SFC conversion, action qualifiers are relevant because some LD constructs do not correspond to simple continuous actions. Set/reset coils, latches, and edge-triggered elements may be better represented through stored, reset, or pulse-like SFC behaviours, depending on the semantics of the original LD program.

2.4.4 Textual Representation of SFC Elements

Although SFC is commonly presented as a graphical language, IEC 61131-3 also defines textual syntax for representing SFC elements. This is relevant for program transformation because SFC structures such as initial steps, steps, transitions, and action associations can also be written in a form that can be parsed or generated by software tools [8, 9]. Lopes [9] identifies four main textual SFC constructs: `INITIAL_STEP`, `STEP`, `TRANSITION`, and `ACTION`. These constructs make it possible to describe the structure of an SFC program without relying exclusively on its graphical notation.

The `INITIAL_STEP` construct defines the initial active step of the chart. A regular step is represented using the `STEP` construct. Both initial and regular steps may contain action associations in their body. An action association connects an action name to the step and may include an action qualifier, which defines how the action behaves while the step is active.

Transitions are represented using the `TRANSITION` construct. A transition specifies the source step or set of source steps, the target step or set of target steps, and the Boolean condition that controls its activation. The transition condition is written as a textual Boolean expression, commonly using ST expression syntax. When the source step or source steps are active and the condition evaluates to `TRUE`, the transition may activate according to the SFC execution rules.

The `ACTION` construct defines a named action separately from the step where it is used. This allows the action body to be declared once and associated with one or more steps through action associations. The body of an action may be written using one of the IEC 61131-3 languages allowed for actions, such as ST or LD, depending on the tool and implementation.

Listing 2.2 presents an example of textual SFC notation using these constructs. The example includes an initial step, a normal step with an action association, two transitions, and an explicitly declared action body.

Listing 2.2: Textual representation of SFC elements using IEC 61131-3-style syntax.

```
INITIAL_STEP Step0:
END_STEP

TRANSITION FROM Step0 TO Step1
  := Start AND NOT Fault;
END_TRANSITION

STEP Step1:
  MotorOn(N);
END_STEP

TRANSITION FROM Step1 TO Step0
  := Stop OR Fault;
END_TRANSITION

ACTION MotorOn:
  Motor := TRUE;
END_ACTION
```

In this example, `Step0` is the initial step and represents the starting point of the chart. The first transition connects `Step0` to `Step1` and fires when the condition `Start AND NOT Fault` is true. The step `Step1` contains the action association `MotorOn(N)`, meaning that the action `MotorOn` is executed with the normal qualifier while `Step1` is active. The second transition returns execution to `Step0` when `Stop OR Fault` becomes true. The `ACTION` block defines the body of the action, which assigns `TRUE` to the variable `Motor`.

Textual SFC syntax is particularly relevant for automatic program transformation because it provides a structured textual representation of the target SFC program. Instead of generating only

graphical objects directly, a conversion tool may first generate textual SFC fragments composed of steps, transitions, action associations, and action bodies. These fragments can then be parsed, analyzed, combined, and finally exported to a graphical or exchangeable representation such as PLCopen XML.

2.4.5 Advantages of SFC for Sequential Behavior

SFC is particularly suitable for representing sequential behavior because it clearly separates three aspects that are often mixed together in LD: the current state of the process, the conditions for changing state, and the actions executed in each state.

This separation improves readability and maintainability, especially in programs where the same behavior would otherwise be encoded in multiple LD rungs. In SFC, the state progression can be inspected directly from the chart structure, whereas in LD it must often be reconstructed by analyzing dependencies between variables across multiple rungs [3, 9].

For this reason, SFC provides a natural target representation for the conversion process addressed in this work. The objective is not to merely rewrite LD Boolean expressions in another language, but to identify the LD structures that encode state-based behavior and represent them as SFC steps, transitions, and actions.

2.5 IEC 61131-3 Development Tools

Automatic analysis of IEC 61131-3 programs often depends on two complementary aspects: an exchange format capable of representing complete projects, and tool support capable of parsing or translating the standardized languages.

This distinction is important for LD-to-SFC conversion. LD and SFC are graphical languages, so their analysis requires access not only to variables and instructions, but also to graphical networks, contacts, coils, steps, transitions, and connections. At the same time, automatic transformation benefits from textual and compiler-level representations, since these can be parsed, traversed, and manipulated more systematically than graphical project files.

2.5.1 PLCopen

PLCopen is an industrial automation organization dedicated to promoting the use of open standards in PLC-based control systems. Its members include companies, tool providers, manufacturers, and educational institutions with an interest in improving compatibility between IEC 61131-3 compliant systems [11].

Beyond the definition of exchange formats, PLCopen has contributed to several areas of industrial control standardization. These include reusable Function Block libraries for motion control, safety-related software specifications, cooperation with the OPC Foundation in the context of OPC UA communication, certification activities, and training programs. These contributions

serve a shared objective: reducing dependence on vendor-specific solutions and improving the portability, reuse, and maintainability of industrial automation software [11].

PLCopen XML was developed to support this very objective: the exchange of IEC 61131-3 project information between compliant development environments. Therefore, PLCopen XML should be understood not only as an XML-based file format, but also as part of a much wider effort to improve interoperability between PLC tools and platforms.

2.5.2 XML-Based Representation and PLCopen XML

PLCopen XML is based on XML (eXtensible Markup Language), which is used to represent structured information through nested elements and attributes. This representation is appropriate for IEC 61131-3 projects because PLC applications are hierarchical: each project contains type definitions, POUs, variable declarations, implementation bodies, graphical networks, and execution instances. Therefore, XML provides a natural way to store both project metadata and program structure [4].

A basic XML fragment is shown in Listing 2.3. The example illustrates the two basic concepts used throughout PLCopen XML: elements and attributes. The element `pou` represents a Program Organization Unit, while the attributes `name` and `pouType` describe properties of that element. Child elements such as `interface` and `body` then store the declaration and implementation parts of the POU.

Listing 2.3: Example of an XML hierarchy using PLC project concepts.

```
<project>
  <types>
    <pous>
      <pou name="program0" pouType="program">
        <interface>
          <!-- variable declarations -->
        </interface>
        <body>
          <!-- program implementation -->
        </body>
      </pou>
    </pous>
  </types>
</project>
```

This example also shows why XML is useful for project exchange. The same file can contain not only code, but also the surrounding information needed to interpret that code. However, XML alone only defines a general syntax for writing structured documents. It does not define which elements are valid for an IEC 61131-3 project, which attributes are required, or where each element

may appear. For that reason, PLCopen XML relies on an XML Schema Definition (XSD), which specifies the valid structure of the document [4].

Listing 2.4 presents an illustrative XSD fragment. It shows how a schema may define that a `pou` element must contain an `interface` and a `body`, and that it must also include attributes such as `name` and `pouType`. In PLCopen XML, the official schema plays this role for the complete IEC 61131-3 project structure.

Listing 2.4: Illustrative XML Schema fragment for a POU-like element.

```
<xs:element name="pou">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="interface"/>
      <xs:element name="body"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="pouType" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

A PLCopen XML document is organized around a root `project` element. This element contains several major parts, including project headers, type definitions, instances, additional data, and documentation. The `fileHeader` stores information about the tool that generated the file, such as company name, product name, product version, and creation date. The `contentHeader` stores project-level information, including the project name and coordinate information for graphical languages such as LD, FBD, and SFC.

Listing 2.5: Adapted example of PLCopen XML project header information.

```
<project>
  <fileHeader
      companyName="ExampleCompany"
      productName="ExampleTool"
      productVersion="1.0"
      creationDateTime="2026-06-18T10:00:00"/>

  <contentHeader name="ExampleProject">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1"/>
      </fbd>
      <ld>
        <scaling x="1" y="1"/>
      </ld>
```

```
      <sfc>
        <scaling x="1" y="1"/>
      </sfc>
    </coordinateInfo>
  </contentHeader>
</project>
```

The `types` element stores data types and POUs. User-defined data types are stored inside `dataTypes`, while POUs are stored inside `pous`. Each `pou` element has a name and a POU type, such as `program`, `functionBlock`, or `function`. Inside each POU, the `interface` element stores the declaration part, including local, input, output, input-output, external, global, or access variables. Each variable is represented by a `variable` element with a name and a type.

Listing 2.6: Adapted example of a POU interface in PLCopen XML.

```
<types>
  <dataTypes/>

  <pous>
    <pou name="program0" pouType="program">
      <interface>
        <inputVars>
          <variable name="Start">
            <type>
              <BOOL/>
            </type>
          </variable>
        </inputVars>

        <localVars>
          <variable name="State0">
            <type>
              <BOOL/>
            </type>
          </variable>
        </localVars>

        <outputVars>
          <variable name="Motor">
            <type>
              <BOOL/>
            </type>
```

```
            </variable>
          </outputVars>
        </interface>
      </pou>
    </pous>
</types>
```

The implementation of a POU is stored inside one or more `body` elements. A body may contain code written in any of the IEC 61131-3 languages. Textual languages such as ST and IL can be represented mainly through textual content. Graphical languages such as LD, FBD, and SFC require structured graphical objects. For this reason, PLCopen XML defines different object groups, including FBD objects, LD objects, and SFC objects [4].

This distinction is essential for LD-to-SFC conversion. An LD input body may contain contacts, coils, Function Block calls, and connections, while an SFC output body must contain steps, transitions, action associations, divergences, convergences, and connections between SFC elements. Therefore, generating an SFC output is not limited to writing transition conditions, the tool must also produce the corresponding graphical object structure for semantics validation.

Listing 2.7: Adapted PLCopen XML-style representation of SFC graphical elements.

```
<body>
  <SFC>
    <step localId="1" name="Step0">
      <position x="100" y="100"/>
    </step>

    <transition localId="2">
      <position x="100" y="180"/>
      <condition>
        <ST>
          Start AND NOT Fault
        </ST>
      </condition>
    </transition>

    <step localId="3" name="Step1">
      <position x="100" y="260"/>
      <connectionPointIn/>
      <connectionPointOut/>
    </step>
  </SFC>
</body>
```

In this type of representation, graphical objects are not interpreted only by their names. They must also be placed in the graphical network and connected to other objects. This is why PLCopen XML is more complex than a textual SFC listing, because it must preserve enough information for a development environment to reconstruct the corresponding graphical SFC.

The `instances` element stores the configuration related information, including configurations, resources, and variables associated with execution contexts. IEC 61131-3 programs are not isolated code fragments. They are executed within the configuration, resource, and task model defined by the standard.

XML, XSD, DOM, and PLCopen XML form the technical foundation of the SFC export process. XML provides the hierarchical serialization model, while the PLCopen XML schema defines the IEC 61131-3-specific elements, attributes, data types, and structural relations that a valid project must contain. DOM-based processing enables these elements to be created and organized as a tree of nodes. Compliance with the PLCopen XML schema therefore determines whether the generated result constitutes an exchangeable IEC 61131-3 project rather than an isolated internal representation [4].

2.5.3 Beremiz and CODESYS

Beremiz and CODESYS are IEC 61131-3 development environments that support graphical and textual PLC languages and the editing, visualization, import, and export of PLC programs [1, 2]. Both can be used to work with any of the 5 standardized languages and support project exchange through PLCopen XML.

Beremiz additionally provides the translation of LD and FBD programs into equivalent ST code. This capability is particularly useful when graphical PLC programs must be converted into a textual representation for subsequent compiler-based analysis. [1]

2.5.4 MATIEC Compiler Architecture

MATIEC is an open-source compiler for the textual representations of the IEC 61131-3 languages such as ST, IL, and SFC [13]. Its processing pipeline is divided into lexical and syntactic analysis, symbol-table construction, semantic analysis, and code generation. Information is progressively added to a shared AST, which is finally consumed by the selected output generator.

The main data flow can be summarized as follows:

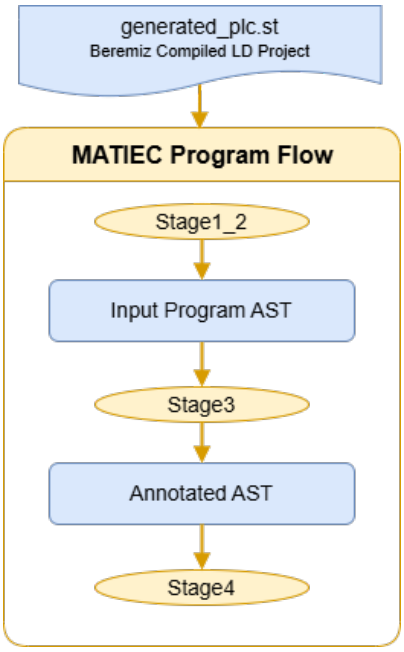


Figure 2.9: MATIEC High-Level Program Architecture and Data Flow.

2.5.4.1 Lexical and Syntactic Analysis

Stages 1 and 2 are implemented together in the `stage1_2` module and execute in a single pass. Stage 1 is the lexical analyser, implemented using Flex. It reads the source characters and groups them into tokens such as identifiers, keywords, operators, literals, and punctuation symbols. The corresponding token values are then supplied to the syntax parser [13].

Stage 2 is implemented using Bison and checks whether the token sequence follows the IEC 61131-3 grammar. During this process, declarations, expressions, assignments, POU definitions, Function Block calls, and SFC constructs are converted into nodes of the MATIEC AST. The node classes are defined in the `absyntax` module.

The lexical analyser and parser exchange information through symbol tables. When the parser encounters declarations of variables, functions, Function Blocks, programs, or derived data types, this information is stored so that subsequent occurrences of the same identifiers can be returned by the lexer as the appropriate token class. This interaction explains why both stages are implemented and executed together in `stage1_2` [13].

After parsing, the resulting AST represents the symbolic organization of the input program. At this point, the tree identifies constructs such as assignments and operators, but semantic properties such as the final data type of an expression may not yet have been resolved.

2.5.4.2 Abstract Syntax Tree Representation

The syntax parser constructs the AST using the classes defined in the `absyntax` module. MATIEC maintains a declarative catalogue of IEC 61131-3 syntax elements in `absyntax.def`. This cata-

logue is included by `absyntax.hh`, where macros generate the corresponding C++ class declarations, child references, constructors, and visitor interfaces [13].

The AST uses three general node classes and seven node forms with zero to six predetermined child references:

- `symbol_c`, the common base class;
- `token_c`, for terminal symbols;
- `list_c`, for nodes containing a variable number of child elements;
- `SYM_REF0` to `SYM_REF6`, for nodes containing between zero and six named child references.

`symbol_c` `symbol_c` is the base class of every node in the MATIEC AST. It records where the corresponding construct appears in the source code, including the file, line, column, and token positions at which it begins and ends. It also stores a reference to its parent, allowing navigation from a child to the enclosing construct.

The class also provides fields that are populated during Stage 3 semantic analysis, including candidate data types, the selected data type, constant-propagation results, declaration references, and a general annotation map. Stage 2 therefore creates the initial AST structure, Stage 3 validates and enriches it with semantic information, and Stage 4 consumes the resulting annotated AST to generate the selected output [13].

`token_c` `token_c` represents terminal symbols produced by the lexical analyser. It inherits the common information from `symbol_c` and adds a `value` field containing the exact text read from the source program.

The token classes are declared with the `SYM_TOKEN` macro. Examples include:

- `identifier_c`, for identifiers such as variable and POU names;
- `integer_c` and `real_c`, for numerical literals;
- `binary_integer_c`, `octal_integer_c`, and `hex_integer_c`, for based integer literals;
- `direct_variable_c`, for directly represented variables;
- `qualifier_c`, for SFC action qualifiers.

For example, the variable name `State_0` is represented by an `identifier_c` whose value contains the character sequence `"State_0"`. Token nodes form the leaves of the AST because they do not contain further syntax elements.

`list_c` `list_c` represents grammar constructs containing a variable number of ordered elements. It stores the current number of children, the allocated capacity, and the references to the elements in the list. Operations are provided to retrieve, locate, append, insert, and remove children [13].

List classes are declared with the `SYM_LIST` macro. Examples include:

- `library_c`, containing the top-level declarations of the parsed source;
- `statement_list_c`, containing an ordered sequence of ST statements;
- `subscript_list_c`, containing the indices of an array access;
- `action_association_list_c`, containing the actions associated with an SFC step;
- `step_name_list_c`, containing the steps connected to an SFC transition;
- `sequential_function_chart_c`, containing the networks of a textual SFC.

Unlike the nodes declared with `SYM_REF0` to `SYM_REF6` macro, a list does not impose a predetermined number of children. A `statement_list_c`, for example, can contain any number of statements while preserving their source order.

`SYM_REF0` to `SYM_REF6` MATIEC uses the macros `SYM_REF0` to `SYM_REF6` for nodes with a fixed number of direct children. The number indicates how many child references the node contains.

Table 2.7 summarizes the seven forms.

Table 2.7: Node forms with a predefined number of child references used by the MATIEC AST.

Macro	Purpose	Examples
SYM_REF0	Represents a symbol with no child nodes.	bool_type_name_c, int_type_name_c, invalid_type_name_c.
SYM_REF1	Represents a symbol with one direct child node.	symbolic_variable_c(var_name), not_expression_c(exp).
SYM_REF2	Represents a symbol with two direct child nodes.	assignment_statement_c(l_exp, r_exp), and_expression_c(l_exp, r_exp), step_c(step_name, action_association_list).
SYM_REF3	Represents a symbol with three direct child nodes.	program_declaration_c(program_type_name, var_declarations, function_block_body), action_association_c(action_name, action_qualifier, indicator_name_list).
SYM_REF4	Represents a symbol with four direct child nodes.	function_declaration_c(function_name, type_name, var_declarations, function_body), if_statement_c(expression, statement_list, elseif_list, else_list).
SYM_REF5	Represents a symbol with five direct child nodes.	transition_c(transition_name, priority, from_steps, to_steps, transition_condition), program_configuration_c(retain_option, program_name, task_name, program_type_name, prog_conf_elements).
SYM_REF6	Represents a symbol with six direct child nodes.	instance_specific_init_c(resource_name, program_name, any_fb_name_list, variable_name, location, initialization).

A SYM_REF0 node differs from a token node since a token retains text read from the source file, whereas a zero-reference node represents a known symbol through its AST class. For example, an occurrence of the elementary type BOOL may be represented by a bool_type_name_c; the node requires no textual child because its class already identifies the type.

The remaining nodes contain explicitly named child references. Some children may be null when the corresponding IEC 61131-3 component is optional. For example, a textual SFC transition may omit its name and priority, but its transition_c node still contains the five fields

defined for that symbol. The fixed references preserve the distinction of each symbol.

A reduced selection of declarations from `absyntax.def` is shown in Listing 2.8.

Listing 2.8: Reduced selection of MATIEC AST node definitions.

```
SYM_TOKEN(identifier_c)
SYM_LIST(library_c)

SYM_REF0(bool_type_name_c)

SYM_REF1(symbolic_variable_c, var_name)
SYM_REF1(not_expression_c, exp)

SYM_REF2(and_expression_c, l_exp, r_exp)
SYM_REF2(or_expression_c, l_exp, r_exp)
SYM_REF2(assignment_statement_c, l_exp, r_exp)

SYM_REF3(program_declaration_c,
         program_type_name,
         var_declarations,
         function_block_body)

SYM_REF5(transition_c,
         transition_name,
         integer,
         from_steps,
         to_steps,
         transition_condition)

SYM_REF6(instance_specific_init_c,
         resource_name,
         program_name,
         any_fb_name_list,
         variable_name,
         location,
         initialization)
```

An `assignment_statement_c`, for example, always contains a left-hand expression and a right-hand expression. These children are accessed through the fields `l_exp` and `r_exp`.

Consider the ST statement in Listing 2.9. Its processed AST hierarchy is represented in Listing 2.10.

Listing 2.9: Example of basic ST expression.

```
Y0 := NOT X0 OR X1;
```

Listing 2.10: Simplified AST hierarchy of Listing 2.9.

```
assignment_statement_c
|__ symbolic_variable_c
|    |__ identifier_c("Y0")
|__ or_expression_c
     |__ not_expression_c
     |    |__ symbolic_variable_c
     |         |__ identifier_c("X1")
     |__ symbolic_variable_c
          |__ identifier_c("X0")
```

The root is an `assignment_statement_c` with two direct children. Its right-hand child contains further fixed number of references nodes, producing a subtree that preserves the Boolean operators, operands, and parenthesized nesting of the original expression.

Visitor-based AST traversal Each AST class implements an `accept` method that dispatches the node to the corresponding visitor operation:

Listing 2.11: Example visitor method used by MATIEC AST nodes.

```
void *accept(visitor_c &visitor) {
    return visitor.visit(this);
}
```

MATIEC uses specific visitor methods for tree iteration, semantic analysis, and code generation [13]. Stage 3 visitors annotate and validate the tree, while Stage 4 visitors process the same node hierarchy to generate the selected output. This separation allows new transformations to reuse the existing parser and semantic analyzer.

2.5.4.3 Semantic Analysis

Stage 3 performs an ordered sequence of semantic checks over the AST. Each check is implemented as a specialized visitor and is invoked through `tree_root->accept(visitor)`. The order is important because several passes depend on annotations produced by earlier analyses [13].

The current execution sequence is:

1. enumeration declaration checking;
2. flow-control analysis;
3. constant propagation and folding;

4. declaration checking;
5. data-type analysis;
6. left-value checking;
7. array-range checking;
8. `CASE` element checking;
9. optional removal of forward dependencies.

Enumeration checking validates enumerated types and their values while flow-control analysis determines the relations between instructions, particularly for IL programs, and provides information required by the following passes.

Constant propagation evaluates expressions whose values can be determined during compilation. The results are stored in the AST and are later used when checking declarations, array bounds, and other constant expressions.

Declaration checking verifies that variables, data types, and array declarations are valid and internally consistent.

Data-type analysis is performed in several steps. The algorithm first determines the possible types of each expression, variable reference, literal, and invocation. It then narrows these candidates to one valid type, reports incompatible expressions, and assigns a final internal type where necessary. The selected type is stored in the corresponding AST node.

The remaining checks verify that assignment targets are writable, array indices are within the declared ranges, and `CASE` alternatives are valid and non-overlapping. When pre-parsing is enabled, the final pass reorders declarations to remove forward dependencies. Otherwise, the original AST order is preserved, which is crucial for LD-to-SFC conversion.

All errors found during Stage 3 are accumulated. If any error remains, compilation stops. Otherwise, the validated and annotated AST is passed to Stage 4.

2.5.4.4 Code Generation

Stage 4 takes the annotated AST as input and converts it into the required output representation. The original MATIEC distribution provides two principal implementations [13]:

- `generate_c`, used by `iec2c` to produce ANSI C;
- `generate_iec`, used by `iec2iec` to regenerate textual IEC 61131-3 code.

Both implementations use visitors to traverse the AST. The compiler Stages 1 and 2 and Stage 3 are therefore independent of the selected output generator: a new Stage 4 can reuse the same lexical analyser, parser, symbol tables, semantic checks, and annotated AST.

The LD-to-SFC compiler follows this architecture by introducing `generate_sfc` as an alternative Stage 4. Its visitors inspect the annotated AST, search for predefined structural patterns,

and construct the corresponding SFC elements. The resulting SFC is subsequently serialized by `generate_sfc_xml` using the PLCopen XML representation.

The transformation is consequently performed after syntax and semantic analysis. Pattern matching operates on resolved AST constructs rather than on unclassified source tokens, while output generation remains isolated from the compiler front-end.

2.6 Existing LD-to-SFC Conversion Approaches

Several research efforts have attempted to extract sequential behavior from LD programs and represent it in SFC form. These approaches differ mainly in the type of information used to recover the sequence: some analyse the graphical and logical structure of the LD program, others introduce a model of the controlled plant, and others explore the state space generated by the program execution. Lopes [9] groups the main contributions into three families: graph-based design recovery, temporal-logic approaches, and state-space exploration.

2.6.1 Graph-Based and Design Recovery Approaches

One of the earliest approaches to LD-to-SFC conversion was proposed by Falcione and Krogh [3]. Their method is based on design recovery from Relay Ladder Logic (RLL), using graph structures to identify dependencies and possible simultaneous execution between rungs. The LD program is analysed in terms of state variables, and two main graph representations are constructed: a dependency graph and a simultaneity graph.

The dependency graph represents precedence relations between state variables. If the activation of one variable depends on the value of another, this relation is represented as a directed connection. The simultaneity graph, on the other hand, represents variables that may be active at the same time. These graphs are then combined and decomposed in order to identify groups of rungs that may correspond to SFC steps or parallel paths.

The main advantage of this approach is that it provides a structural method for recovering a possible SFC organization from LD programs. However, because the method is primarily based on graph relationships between rungs and state variables, it may not fully capture the semantic meaning of the sequential behavior encoded in the LD program. In particular, implicit memory structures, feedback contacts, and process-dependent behaviors may require additional interpretation beyond the graph decomposition itself [9].

2.6.2 Temporal-Logic and Plant-Model Approaches

Zanma *et al.* [15] proposed a different approach, based on the idea that the sequence of operations cannot always be recovered from the LD program alone. In many industrial systems, the order in which actions occur depends not only on the controller logic, but also on the response of the physical process. For example, a valve may be opened by the controller, but the next action may

only occur after a sensor confirms that the tank has reached a certain level. In this case, the LD program and the plant behaviour must be analysed together.

For this reason, the method models the closed-loop system, composed of the LD controller and the controlled plant. Temporal logic is then used to describe relations between controller actions, plant events, and sensor feedback. This allows the method to infer ordering relations between actions and to build an SFC that reflects the expected sequence of the physical process.

The main advantage of this approach is that it can recover sequences that are not explicit in the LD structure itself. Since the plant model provides information about how the process reacts to controller outputs, the generated SFC can better reflect the real execution order of the system.

However, this is also its main limitation. The method requires a model of the controlled plant, which may not be available when analysing legacy PLC programs. Creating such a model can be difficult, especially for large industrial systems or poorly documented installations. Therefore, although temporal-logic approaches provide a rigorous way to recover process-dependent sequences, their practical application is limited by the need for external plant knowledge and by the complexity of modelling the physical system [9].

2.6.3 State-Space Approaches

State-space approaches attempt to recover the implicit sequence of an LD program by analysing the possible combinations of input, output, and internal variable values. Nandhan and Babu [10] proposed a method in which the LD program is evaluated repeatedly while input variables are toggled, generating a state space that represents possible program behaviours. Heuristic rules are then applied to remove infeasible states and derive a reduced model from which an SFC can be constructed.

The main benefit of this type of approach is that it can reveal hidden sequential behaviour without requiring the programmer to explicitly identify states in the LD program. Since LD often encodes state using internal variables, feedback paths, and set/reset logic, a state-space representation can expose relationships that are not immediately visible in the graphical program.

Nevertheless, state-space approaches are affected by scalability problems. As the number of input and internal variables increases, the number of possible states grows exponentially. This can make the method computationally expensive for larger industrial programs. In addition, the quality of the result depends strongly on the heuristic rules used to remove infeasible or irrelevant states. If these rules are too aggressive, valid behaviours may be removed; if they are too permissive, the resulting state space may remain too complex to be converted into a readable SFC [9].

2.6.4 Lopes' Optimization of State-Space Approaches

Lopes [9] builds on the state-space approach and proposes an improved methodology for converting LD programs into SFC. His work uses PLCopen XML as the input format, extracts an internal

representation of the LD program, and generates a state space that represents the implicit sequential behavior encoded in the original LD logic. The state space is then reduced using heuristic rules, and the resulting structure is used to build an SFC representation.

A relevant contribution of Lopes' work is the distinction between the graphical LD structure and the execution semantics of the program. The LD program is not treated merely as a set of Boolean equations; instead, the method accounts for the cyclic execution of PLC programs and for the role of memory-dependent constructs such as feedback and self-holding logic. This is particularly important because these constructs are often responsible for encoding the state-like behavior that must later be represented as SFC steps and transitions.

Despite these contributions, the approach still depends on state-space generation and reduction. This means that the scalability issue is not completely eliminated, even if heuristic rules reduce the number of states that need to be considered.

For the present work, Lopes' dissertation is especially relevant because it establishes the main problem, analyses previous LD-to-SFC approaches, and demonstrates that implicit sequential behavior can be recovered from LD programs. However, the approach developed in this dissertation follows a different direction: instead of relying primarily on state-space generation, it investigates the use of ST and AST-based structural pattern detection as an intermediate step for identifying LD constructs that can be mapped into SFC.

2.6.5 Comparative Analysis

The approaches discussed above show that LD-to-SFC conversion can be addressed from different perspectives. Graph-based methods focus on structural relations between rungs and variables. Temporal-logic methods introduce plant behavior in order to recover action ordering. State-space methods analyse possible program states and transitions generated by the LD execution semantics. Lopes' work improves the state-space direction by introducing heuristic rules and a prototype based on PLCopen XML.

However, none of these approaches fully solves the problem of automatic, scalable, and semantically clear LD-to-SFC conversion. Graph-based methods may miss implicit memory semantics, temporal-logic methods require external plant models, and state-space methods suffer from combinatorial growth. These limitations motivate the exploration of a pattern-oriented approach, where recurring LD structures are detected directly in an intermediate representation and mapped to SFC constructs.

Chapter 3

Methodology

This chapter presents the methodology adopted for the automatic conversion of selected IEC 61131-3 Ladder Diagram programs into Sequential Function Chart representations. The proposed approach follows a compiler-oriented pipeline based on the transformation of graphical LD logic into Structured Text, the construction of Abstract Syntax Trees, rule-driven pattern matching, and the generation of SFC code in PLCopen XML format.

The methodology is based on the observation that LD programs used to encode sequential behavior often rely on a limited set of recurring programming patterns. These patterns are used to represent implicit states, activation conditions, deactivation conditions, feedback memory, and state-dependent actions. Once such structures are detected, they can be translated into equivalent SFC constructs, where the same behavior is represented through steps, transitions, and actions.

Fig. 3.1 presents an overview of the proposed LD-to-SFC conversion pipeline. Beremiz is used to edit or import the LD program and translate it into ST. The generated ST program and the pattern definition file are then processed by the modified MATIEC-based tool. Both inputs are parsed into AST representations, which are used by the pattern matching module to generate the corresponding SFC program in PLCopen XML format.

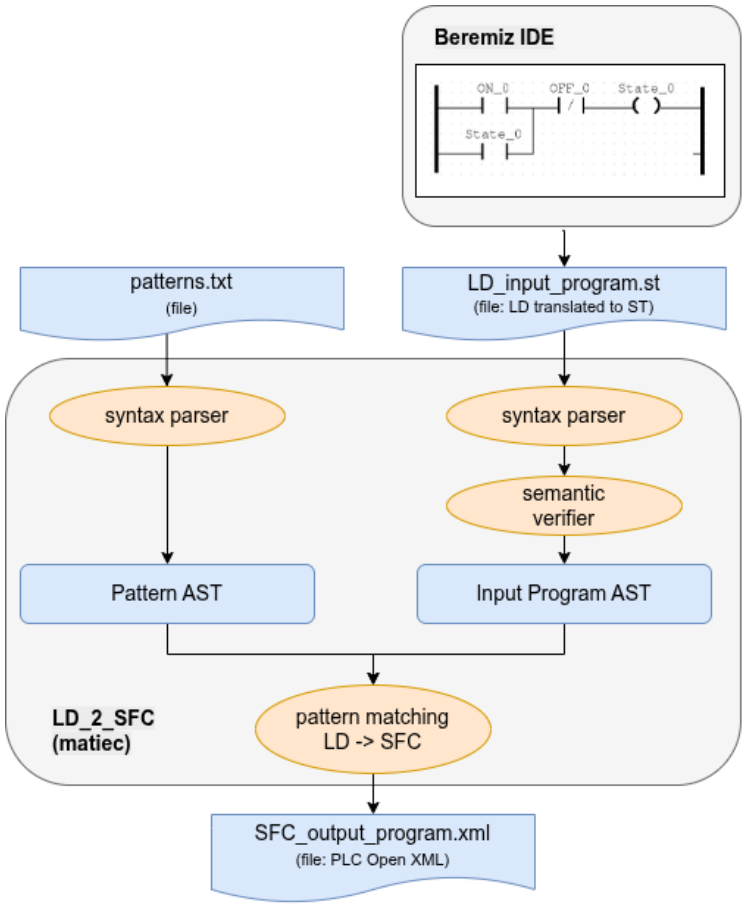


Figure 3.1: Overview of the proposed LD-to-SFC conversion pipeline. Reproduced from [14].

3.1 Problem Definition and Context

The problem addressed in this dissertation is the automatic conversion of LD programs that encode state-machine-like behaviour into equivalent SFC programs. LD is widely used in industrial automation due to its similarity to electrical relay diagrams and its familiarity among PLC engineers and technicians. However, sequential behaviour in LD is often encoded implicitly through Boolean expressions, feedback contacts, internal variables, set/reset coils, and rung execution order, making the underlying state structure difficult to identify.

SFC represents sequential behaviour explicitly. Steps represent control states or phases, transitions define the conditions for changing state, and actions describe the behaviour associated with active steps. Therefore, converting LD into SFC requires recovering the implicit state structure of the LD program and representing it through SFC elements.

The proposed approach does not enumerate the complete program state space. Instead, it searches for recurring LD programming patterns commonly used to encode sequential logic. When a pattern is detected, it is translated into an equivalent SFC structure according to a predefined transformation rule.

The methodology is implemented as a compiler-oriented toolchain. Beremiz is used to edit or import LD programs and translate graphical LD logic into ST. MATIEC is then reused and extended to parse the generated ST code, build the corresponding AST, parse the pattern definitions, perform AST-level pattern matching, and generate the SFC output.

3.2 LD Representation through Structured Text

The first stage of the methodology consists of transforming the graphical LD program into a textual IEC 61131-3 representation. This step is necessary because the proposed conversion tool operates over compiler-level structures, namely Abstract Syntax Trees, rather than directly over graphical LD objects.

In a PLCopen XML project, an LD program is represented through graphical elements such as contacts, coils, branches, connections, positions, and coordinates. Although this information is sufficient to reconstruct the graphical diagram, direct analysis of this representation would require implementing a dedicated LD graph parser. Instead, the proposed toolchain uses Structured Text as an intermediate representation.

Beremiz is used to translate the graphical LD program into equivalent ST code. This conversion transforms LD contact networks and output elements into textual assignments and Boolean expressions that can later be parsed by MATIEC. Contacts placed sequentially in the same path are translated as `AND` operations, alternative branches are translated as `OR` operations, and normally closed contacts are translated as negated variables. The output element associated with the rung determines the assignment or update structure generated in ST.

Each translation depends on the LD graphical elements involved. Contacts, branches, coils, Function Blocks, set/reset elements, and edge-sensitive elements may lead to different ST structures. Therefore, the LD-to-ST stage must preserve the relevant semantics of each element, ensuring that the resulting AST exposes the assignments, Boolean expressions, feedback dependencies, state updates, and function calls required for pattern matching.

For example, the LD self-holding pattern shown in Fig. 3.2 encodes a single state using an activation condition, a deactivation condition, and a feedback contact. Beremiz translates this structure into the following ST assignment:

```
State_0 := NOT OFF_0 AND (ON_0 OR State_0);
```

In this expression, `ON_0` represents the activation condition, `OFF_0` represents the deactivation condition, and `State_0` represents the memory variable used to retain the active state. The term `ON_0 OR State_0` keeps the state active after activation, while `NOT OFF_0` disables the state when the deactivation condition becomes true.

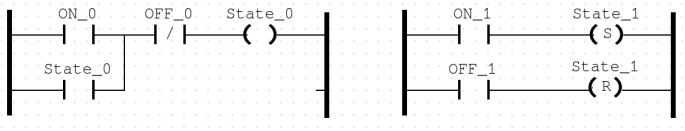


Figure 3.2: Common LD patterns used to encode a state with activation and deactivation conditions.

The presence of `State_0` on both sides of the assignment is the key structural feature of this pattern. It shows that the next value of the state variable depends on its previous value. This feedback dependency is not represented as an explicit state in LD, but it becomes visible in the ST expression and can later be detected in the AST.

The result of this stage is an ST program that preserves the logical dependencies required for conversion while making them accessible to compiler tools. This ST program is then used as the input for the AST construction stage.

3.3 AST Representation of the Input Program

After the LD program is translated into ST, the resulting code is processed by the MATIEC compiler front-end. In its original workflow, MATIEC parses IEC 61131-3 textual languages and generates equivalent C code. In this work, the lexical analysis, syntax analysis, and semantic verification stages are reused, while the original C code generation stage is replaced by the LD-to-SFC transformation module.

The result of the parsing stage is an Abstract Syntax Tree (AST) representing the structure of the input ST program. In this tree, assignments, Boolean operators, symbolic variables, Function Block calls, and other IEC 61131-3 constructs are represented as nodes that can be traversed by the conversion algorithm.

For example, the ST assignment generated from the LD self-holding pattern is:

```
State_0 := NOT OFF_0 AND (ON_0 OR State_0);
```

The corresponding AST has an assignment statement as its root node. The left-hand side is the symbolic variable `State_0`, while the right-hand side is a Boolean expression composed of an `AND` operation. This operation combines the negated deactivation condition, `NOT OFF_0`, with the self-holding expression, `ON_0 OR State_0`.

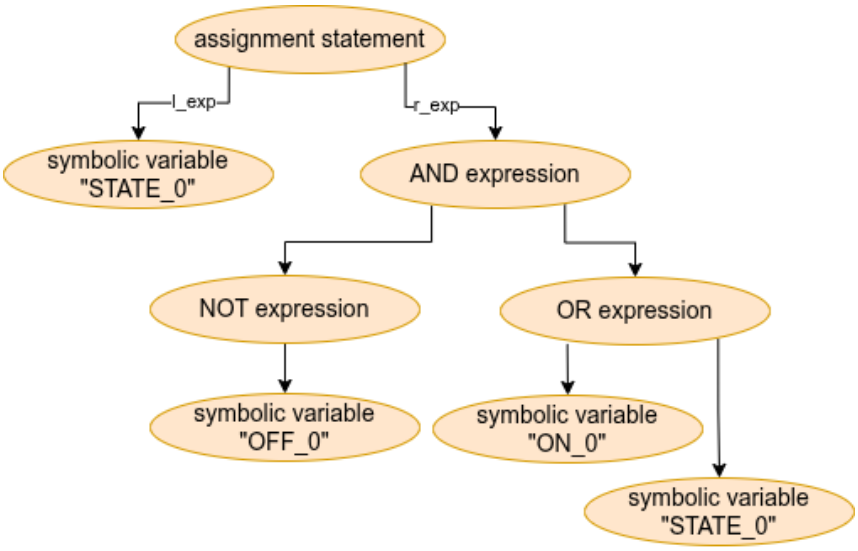


Figure 3.3: AST representation of the ST assignment generated from an LD self-holding pattern.

This representation makes the structure of the LD-derived logic explicit. The feedback dependency is visible because `State_0` appears both as the assignment target and inside the Boolean expression that determines its next value. This is the structural feature that allows the pattern matching algorithm to recognize the expression as a state-encoding construct.

Using an AST also avoids simple text-based matching. The algorithm does not search for exact strings, but for equivalent syntactic structures. Therefore, a pattern can still be detected when the activation or deactivation conditions are replaced by more complex Boolean expressions, as long as the overall structure of the pattern is preserved.

3.4 Pattern Definition and AST-Based Pattern Matching

The conversion tool receives a second input file, named `patterns.txt`, containing the patterns to be searched and the corresponding SFC structures to be generated. Each pattern is defined as a pair composed of an input and output pattern:

$$\{pattern_IN,\ pattern_OUT\}.$$

The input pattern describes the ST structure to be detected in the input program, while the output pattern describes the SFC fragment that should be generated when the pattern is found. Since both the input program and the pattern file are parsed into ASTs, the matching process compares tree structures rather than just text strings.

A top-level pattern is used to define which subpatterns should be searched in the input program. In the current implementation, this is done with the `PATTERN_N__` construct, which instructs the pattern matching algorithm to search for any number of occurrences of the referenced subpatterns. For example, `PATTERN_N__(p1, p2, p3)` means that only patterns `p1`, `p2`, and `p3` are considered at that point, and that each of them may be matched any number of times.

Listing 3.1: Top-level pattern used to search for LD-to-SFC subpatterns.

```
pattern_IN__ AAA
  VAR
    x : BOOL;
  END_VAR

  PATTERN_N__(p1, p2, p3)
end_pattern__


pattern_OUT__ AAA
  VAR
    x : BOOL;
  END_VAR

  INITIAL_STEP Step0:
  END_STEP

  PATTERN_N__(p1, p2, p3)
end_pattern__
```

The input pattern of the top-level pattern pair defines the set of subpatterns to be searched in the input AST. The output pattern defines the initial SFC structure to be generated before the matched subpatterns are expanded. In this case, the output pattern creates an initial step, `Step0`, followed by a `PATTERN_N__` construct that marks where the SFC fragments generated by the matched subpatterns may be inserted.

Fig. 3.4 represents the internal AST generated from the output side of the top-level pattern. The main SFC node contains an SFC network. Inside this network, the initial step is represented by the `initial_step_c` node and identified as `Step0`. The `match_n_pattern_list_c` node stores the references to the subpatterns that may be expanded at this point of the generated SFC.

When a referenced subpattern is matched, its output pattern is expanded at the position represented by the `match_n_pattern_list_c` node, generating additional SFC nodes, such as steps, transitions, and action associations, which are handled by the post-processing stages described later.

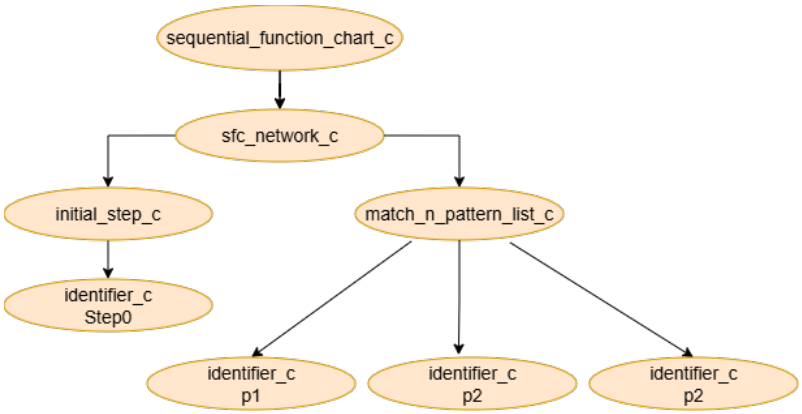


Figure 3.4: Simplified AST representation of the top-level output pattern. The SFC network is initialized with `Step0`, while the `PATTERN_N__` node identifies the subpatterns whose generated SFC fragments may be inserted into the network.

The subpatterns define the actual LD-to-SFC transformations. For example, pattern `p2` represents an LD structure where a Boolean variable is used with feedback, meaning that its next value depends on its own previous value. This feedback is what characterizes the construct as a state-encoding pattern. This expression also includes a parent-state dependency, an activation condition, and a deactivation condition, as shown in Fig. 3.5.

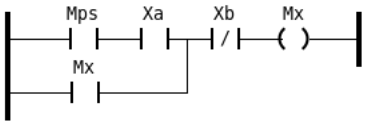


Figure 3.5: LD state-encoding structure represented by pattern `p2`.

The LD structure in Fig. 3.5 is translated into the ST assignment used in the input pattern of the state-encoding subpattern:

Listing 3.2: Example of input pattern of a state-encoding subpattern.

```
pattern_IN__ p2
  VAR
    Mx  : BOOL;
    Mps : BOOL;
    Xa  : BOOL;
    Xb  : BOOL;
  END_VAR

  ST_PART_BEG__
    Mx := NOT Xb AND (Mx OR (Mps AND Xa));
  ST_PART_END__
end_pattern__
```

In this pattern, `Mx` is the state variable being detected, `Mps` represents the previous or parent state condition, `Xa` is the local activation condition, and `Xb` is the deactivation condition. The term `Mx` provides the feedback required to keep the state active after activation. The term `Mps AND Xa` defines when the state may become active, and `NOT Xb` disables the state when the exit condition becomes true.

The corresponding output pattern generates an SFC fragment composed of an entry transition, a step, and an exit transition:

Listing 3.3: Example output pattern of the state-encoding subpattern.

```
pattern_OUT__ p2
  VAR
    Mx  : BOOL;
    Mps : BOOL;
    Xa  : BOOL;
    Xb  : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := Xa AND Mps;
  END_TRANSITION

  STEP StepTo:
    Mx(N);
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := Xb;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

Taken in isolation, this subpattern generates the SFC fragment shown in Fig. 3.6. After an appropriate match, this subpattern converts an LD snippet that encodes a single state into an equivalent SFC snippet composed of one step and two transitions. The feedback assignment is represented by the SFC step `StepTo`. The transition leading to that step is guarded by `Xa AND Mps`, reflecting both the local activation condition and the dependency on the previous or parent state. The exit transition is guarded by `Xb`, which corresponds to the deactivation condition of the state itself.

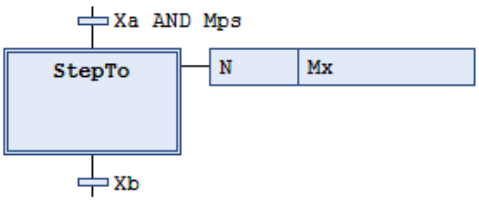


Figure 3.6: SFC fragment generated directly from subpattern `p2`.

At this stage, the parent-state dependency `Mps` is still present in the transition condition. Later, when this fragment is integrated into the complete SFC, this dependency may be represented structurally by the connection from the source step `StepFrom`. In that case, `Mps` can be removed from the transition condition, since the transition is only evaluated when the preceding SFC step is active.

The variables declared inside the pattern act as placeholders. During pattern matching, each placeholder is matched with a corresponding element in the input AST, respecting type compatibility. Once a match is accepted, the tool stores a binding between each placeholder and the concrete variable found in the input program.

If the same placeholder appears more than once in a pattern, all its occurrences must refer to the same matched variable. This ensures that the structural relation expressed by the pattern is preserved during replacement.

The advantage of AST-based matching is that the comparison is performed over the structure of the expression, not over its textual formatting. For example, in the pattern expression `Xa AND Mps`, the algorithm searches for an `AND` expression with two compatible Boolean operands. If the input program contains `X1 AND M0`, the placeholder `Mps` is matched with `M0` and the placeholder `Xa` is matched with `X1`. These mappings are then used to replace the placeholder names in the output pattern with the concrete variable names found in the input program.

Other patterns follow the same principle but capture different LD structures. For instance, while `p2` captures a state-encoding assignment with feedback, `p3` captures a simple Boolean assignment without feedback:

Listing 3.4: Input pattern of a simple action assignment.

```
pattern_IN__ p3
  VAR
    dd : BOOL;
    s2 : BOOL;
  END_VAR

  ST_PART_BEG__
    s2 := dd;
  ST_PART_END__
end_pattern__
```

Unlike the state-encoding pattern, this assignment does not contain feedback from the assigned variable. The variable `s2` is assigned directly from `dd`, and its next value does not depend on its previous value. Therefore, this pattern does not define a new state. Instead, it represents action logic associated with an already identified state or step.

Listing 3.5: Output pattern of a simple action assignment.

```
pattern_OUT__ p3
  VAR
    dd : BOOL;
    s2 : BOOL;
  END_VAR

  SFC_PART_BEG__
  STEP StepOn:
    s2(N);
  END_STEP
  SFC_PART_END__
end_pattern__
```

The generated SFC fragment associates the action `s2` with the step `StepOn`, using the normal action qualifier `N`. During post-processing, this action association is connected to the step corresponding to the condition represented by `dd`.

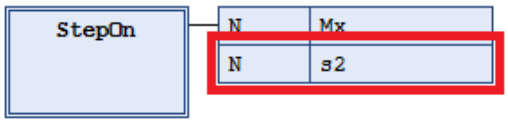


Figure 3.7: SFC fragment generated indirectly from subpattern `p3`.

After the patterns are defined, the matching algorithm searches for their occurrences in the input program AST. The traversal starts at the root of the input AST and proceeds through its intermediate and leaf nodes. At each position, the subtree rooted at that node is compared with the stored `pattern_IN` ASTs.

A match is accepted when the compared structures are compatible. This includes checking the type of AST nodes and the data types of the matched elements.

When a match is found, the tool records the matched pattern, the corresponding AST location, and the mappings between pattern placeholders and concrete input variables. Additional information is also extracted from each match, such as state variables, activation conditions, deactivation conditions, right-hand-side variables, parent-state dependencies, and guard expressions. This information is stored as match annotations.

These annotations conclude the initial pattern matching stage. At this point, the tool has identified the relevant LD-derived structures in the input AST and generated the corresponding

local SFC fragments. However, these fragments are not yet a complete SFC program. State-encoding patterns, such as `p2`, generate step and transition fragments, while action-generating patterns, such as `p3`, generate action candidates. Therefore, the final SFC cannot be produced by simply placing isolated pattern outputs side by side. Additional post-processing is required to analyse the annotated dependencies between matches, associate actions with the correct steps, aggregate compatible transitions, and generate the final PLCopen XML output.

3.5 Post-Processing and PLCopen XML Export

The previous section described how individual LD-to-SFC patterns are detected and how their corresponding SFC fragments are generated. However, the result of pattern matching is not yet a complete SFC program. At this point, the generated elements remain local fragments, such as steps, entry and exit transitions, and action candidates. These fragments must be analyzed and combined before a coherent SFC can be exported in PLCopen XML format.

Figure 3.8 presents the summarized internal processing pipeline of the LD-to-SFC conversion module. After pattern search and matching, two main structural post-processing stages are applied: SFC aggregation and action association. Once these stages have produced the complete SFC AST, the resulting structure is serialized as a PLCopen XML project.

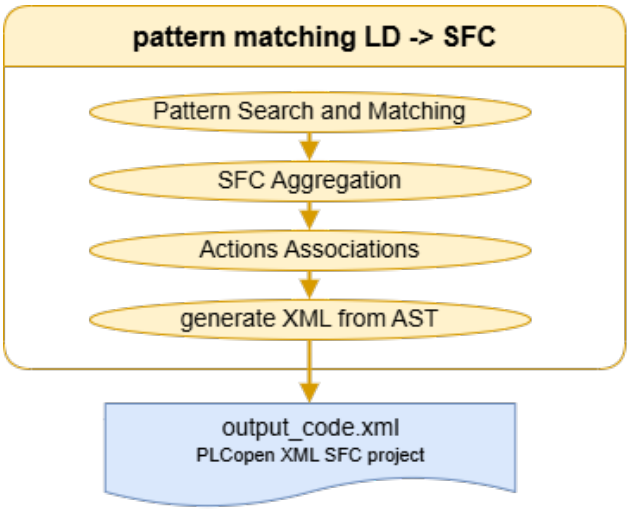


Figure 3.8: Internal processing stages of the LD-to-SFC conversion module.

The post-processing stages use the match annotations collected during pattern matching, including the detected state variables, activation and deactivation conditions, right-hand side variables, state dependencies, guard expressions, and original statement positions.

The first post-processing stage is SFC aggregation. This stage resolves the dependencies between the detected state variables, assigns concrete SFC steps, merges the locally generated fragments, and resolves their entry and exit transitions. It also determines whether the resulting connections form simple sequences, selection divergences, or simultaneous divergences.

Two transitions are considered compatible when the entry transition of one fragment corresponds to the exit transition of another fragment. In practice, this means that the transition conditions must share the same activation condition, with the entry transition possibly including an additional `AND` condition representing the state of the preceding step. When this dependency becomes explicit through the SFC structure and chart flow, the redundant state check can be removed from the transition condition, leaving just the matched activation condition.

For example, a local fragment generated from a state-encoding pattern may initially contain an entry transition guarded by `Xa AND Mps`. After aggregation, if `Mps` is represented structurally by the preceding SFC step, the transition condition can be reduced to `Xa`. This is because the transition is only evaluated when its source step is active, so the parent-state dependency is already enforced by the SFC execution semantics.

The transition aggregation algorithm must also preserve the execution semantics of the original LD program. In LD, rungs are evaluated sequentially, from top to bottom, and assignments performed in earlier rungs may affect the evaluation of later rungs. For this reason, transitions cannot be merged only on the basis of similar Boolean conditions. The aggregation stage must also consider the original rung order, since this order constrains which transitions may be merged without changing the program behaviour.

Besides creating simple sequential connections, transition aggregation must also identify cases where the generated SFC branches into more than one path. This occurs when one detected state may lead to several following steps. In such cases, the algorithm must determine whether the branch represents a selection or a simultaneous divergence.

A selection divergence represents alternative execution paths. Several transitions leave the same source step, each with its own condition, and only the branch whose condition becomes true is followed. A simultaneous divergence, on the other hand, represents the parallel activation of several branches. In this case, multiple target steps become active together, and the divergence is not interpreted as a choice between alternatives.

Consider the following assignments generated from the original LD program. Their order is preserved in the ST program supplied to MATIEC.

```
M4 := NOT X5 AND (M4 OR (M3 AND X4));
M3 := NOT X4 AND (M3 OR (M1 AND X2));
M2 := NOT X3 AND (M2 OR (M1 AND X2));
M1 := NOT X2 AND (M1 OR (M0 AND X1));
M0 := NOT X1 AND (M0 OR X0);
```

Each assignment describes the activation and deactivation of one state variable. The state dependency and transition condition extracted from each statement are summarized in Table 3.1.

Table 3.1: State dependencies extracted from the generated ST program.

State	Predecessor	Activation condition	Deactivation condition
M0	Initial step	X0	X1
M1	M0	X1	X2
M2	M1	X2	X3
M3	M1	X2	X4
M4	M3	X4	X5

The first merge occurs between M0 and M1. The deactivation condition of M0 is X1, while the activation of M1 depends on M0 AND X1. These expressions describe the same state change and are therefore represented by a single transition from the step associated with M0 to the step associated with M1.

The same process connects M3 to M4 through X4. However, both M2 and M3 depend on M1 and use X2 as their activation condition, so when the step associated with M1 is active and X2 becomes true, both states are activated. The tool represents this case through a transition followed by a simultaneous divergence, as shown in Figure 3.9.

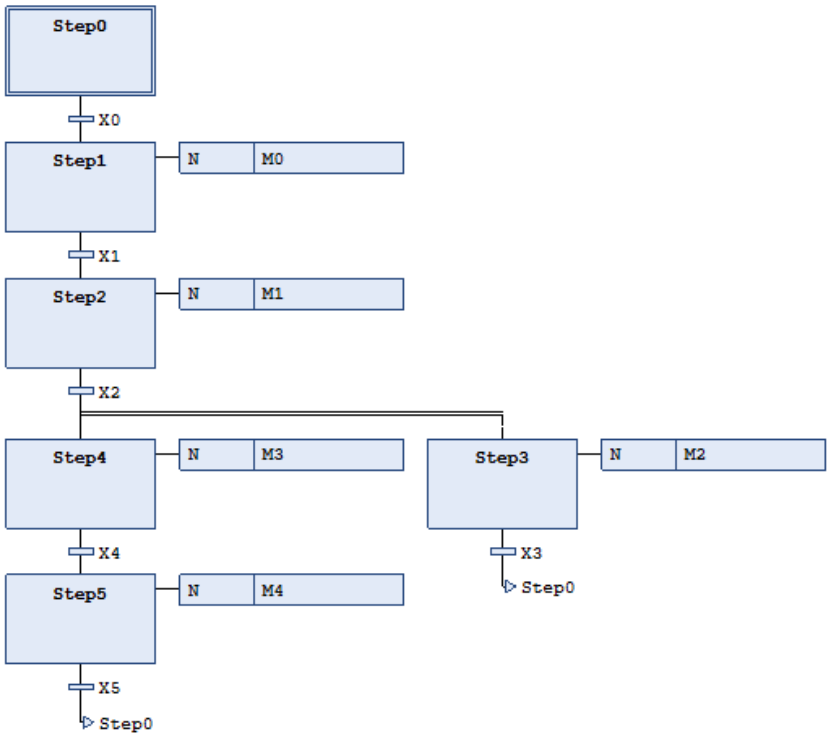


Figure 3.9: SFC generated from the state assignments, including the simultaneous divergence from the step associated with M1.

The original assignment order is also relevant to the classification of this divergence. In the generated ST program, the assignments to M3 and M2 are evaluated before the assignment that

resets `M1`. Consequently, both expressions read the pre-update value of `M1` and may activate their respective states during the same execution cycle. The tool therefore creates a common transition with condition `X2`, followed by a simultaneous divergence to the steps associated with `M3` and `M2`. Since the assignment to `M3` occurs first, its branch is generated first.

Finally, no successor state depends on `M2 AND X3` or `M4 AND X5`. Their deactivation conditions therefore produce return transitions to the initial step.

Therefore, transition aggregation is not limited to merely merging compatible entry and exit transitions. It also analyses source steps, target steps, transition conditions, rung order, and detected dependencies in order to decide whether the resulting SFC structure should be represented as a simple sequence, a selection divergence, or a simultaneous divergence.

The second structural post-processing stage is action association. This stage is performed after SFC aggregation because the concrete step structure must first be established. Once the detected states and transitions have been combined into a coherent SFC, the tool can determine the steps to which the actions generated from action patterns must be attached.

An action pattern such as `p3` may generate an action candidate, for example `s2(N)`. However, the isolated match does not by itself determine the step or steps with which that action must be associated. This association is determined by analyzing the dependency between the action condition and the state variables detected in other matches. Therefore, the same action may be associated with more than one step when the corresponding LD logic is enabled in more than one detected state.

After transition aggregation and action association, the resulting SFC AST structure is converted into PLCopen XML. This output format allows the generated SFC program to be imported into IEC 61131-3 compliant development environments, such as Beremiz or CODESYS.

The final result of the pipeline is not only an internal SFC representation, but also an exchangeable PLCopen XML file that can be opened, inspected, and further edited in existing industrial automation tools.

Chapter 4

Development of the LD-to-SFC Conversion Tool

This chapter describes the implementation of the LD-to-SFC conversion methodology presented in Chapter 3. The approach was implemented by extending MATIEC with a new Stage 4 module that identifies LD-derived patterns in an ST program and generates the corresponding SFC structure.

The conversion process uses Beremiz to translate the original LD program into ST. The modified MATIEC compiler parses and analyzes the ST code, applies the conversion patterns to its Abstract Syntax Tree (AST), and exports the generated SFC as a PLCopen XML project.

The following sections present the tool requirements, software architecture, MATIEC integration, pattern-matching algorithm, SFC generation and post-processing, and PLCopen XML export.

4.1 Tool Requirements and Design Decisions

The implementation requirements were derived from the conversion methodology presented in Chapter 3. The main requirements are:

1. The input shall be an ST representation generated from an LD program.
2. The existing MATIEC lexical, syntactic, and semantic analysis stages shall be reused.
3. The conversion shall operate on the semantically analyzed MATIEC AST.
4. Conversion patterns shall be defined in an external file, allowing new patterns to be added without modifying the matching algorithm.
5. Identifier placeholders captured during matching shall be preserved and substituted into the generated SFC fragments.
6. The tool shall generate the required SFC elements, including steps, transitions, actions, divergences, and convergences.

7. The generated SFC shall be exported in PLCopen XML format for import into a compatible IEC 61131-3 development environment.

8. Parsing errors, missing pattern definitions, and conversion errors detected by the implementation shall be reported during execution.

Reuse of the MATIEC Analysis Pipeline Instead of implementing a new parser and semantic analyzer, the tool reuses the existing MATIEC analysis pipeline. MATIEC converts the ST source program into an AST and performs declaration checking, type analysis, constant propagation, and other semantic validations.

The conversion algorithm therefore operates on a validated and annotated AST. It also reuses the existing MATIEC node classes for assignments, expressions, variable references, Function calls, Function Block calls, and SFC elements.

Implementation as a New Stage 4 The conversion logic was implemented as a new MATIEC Stage 4 module named `generate_sfc`. It receives the annotated AST produced by the preceding analysis stages and transforms the recognized ST structures into SFC elements.

Implementing the conversion at this stage avoids changing how ordinary ST declarations, expressions, and POUs are analyzed. The lexical, syntactic, and semantic analysis stages remain unchanged for the input program.

Small extensions were added to the lexical analyzer and parser to recognize the delimiters and constructs used in the external pattern files. These extensions create the corresponding pattern ASTs, while the matching and transformation operations are performed by `generate_sfc`.

External Pattern Definitions and Output Format The conversion patterns are defined in a separate file from the C++ implementation. Each pattern contains an input structure to be matched in the source AST and an output structure describing the corresponding SFC fragment to be generated.

This separation allows new patterns to be added or existing patterns to be modified without changing the tree-matching algorithm. It also makes each supported conversion rule independent and individually testable.

The generated SFC is exported as PLCopen XML rather than only as textual IEC 61131-3 code. PLCopen XML can represent the graphical structure of an SFC, including steps, transitions, action blocks, connections, and divergences.

The output file can therefore be imported into a compatible development environment, where the generated chart can be displayed, inspected, and edited graphically.

4.2 Tool Architecture

The developed conversion process is implemented as a new MATIEC Stage 4. At this point, the input LD program has already been translated into ST and parsed into an annotated input-program

AST, while the external pattern file has been parsed into a pattern AST. These two AST representations form the input of the implemented LD-to-SFC conversion stage, as shown in Figure 4.1.

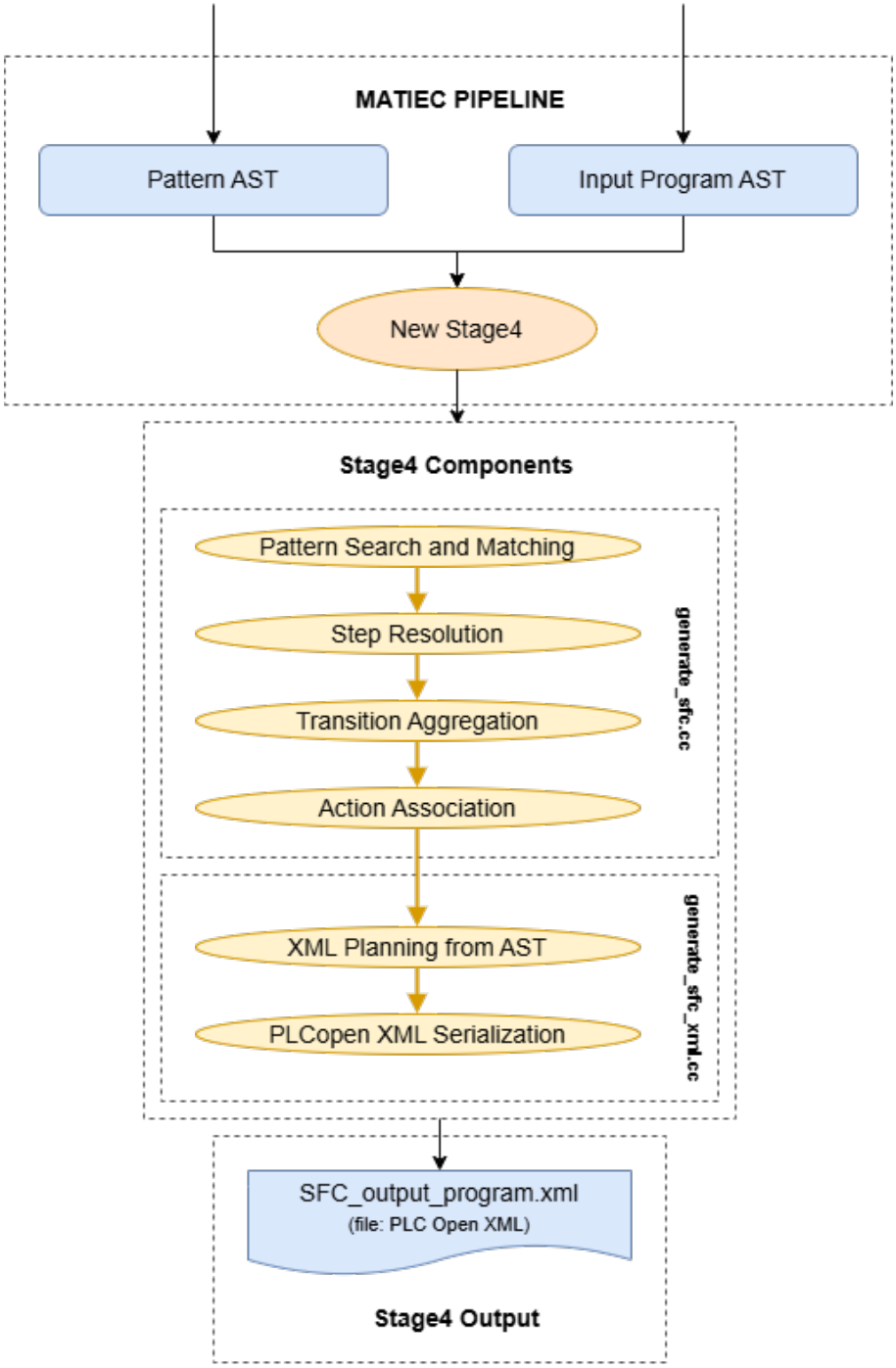


Figure 4.1: Program flow of the implemented Stage 4 LD-to-SFC conversion modules.

The transformation is divided into two modules. The `generate_sfc.cc` module performs the AST-based pattern matching and builds the intermediate SFC AST. This includes matching the

input-program AST against the registered patterns, resolving the generated step structure, aggregating compatible transitions, and associating actions with the corresponding steps.

The resulting SFC AST is then processed by `generate_sfc_xml.cc`. This module prepares the XML-level representation of the chart, including identifiers, graphical connections, divergences, convergences, jumps, transitions, steps, and action blocks, and serializes the final result as a PLCopen XML project.

4.3 `generate_sfc` Module

The main conversion procedure is implemented in `generate_sfc.cc` as a MATIEC Stage 4 module. It operates on the annotated AST of the input ST program and on the input- and output-pattern ASTs obtained from the external pattern file.

4.3.1 Module Organisation and Execution Flow

The `generate_sfc.cc` module is implemented using the visitor-based structure already used internally by MATIEC. In this context, a visitor is an object that traverses the AST and performs a specific operation when visiting each node. Instead of embedding all processing logic directly in the AST node classes, each visitor defines the behavior required for a particular task, such as registering patterns, comparing subtrees, duplicating AST fragments, replacing identifiers, or constructing the output SFC AST.

Listing 4.1 illustrates, in simplified C++, how the visitor mechanism is used in MATIEC. Each AST node declares an `accept` method, which receives a visitor object. The visitor then executes the `visit` method associated with the concrete type of node. Visitors derived from `iterator_visitor_c` usually continue the traversal through the child nodes, while other specialized visitors override the relevant `visit` methods to perform a specific operation.

Listing 4.1: Simplified illustration of MATIEC visitors.

```
class symbol_c {
public:
    virtual void *accept(visitor_c &visitor) {
        return visitor.visit(this);
    }
};

class iterator_visitor_c : public visitor_c {
public:
    void *visit(assignment_statement_c *symbol) {
        symbol->l_exp->accept(*this);
        symbol->r_exp->accept(*this);
        return NULL;
```

```
    }
};

class match_visitor_c : public visitor_c {
private:
    symbol_c *search_pattern;

public:
    void *visit(assignment_statement_c *symbol) {
        assignment_statement_c *pattern =
            dynamic_cast<assignment_statement_c *>(search_pattern);

        if (pattern == NULL)
            return NULL;

        // Compare left-hand side
        search_pattern = pattern->l_exp;
        if (symbol->l_exp->accept(*this) == NULL)
            return NULL;

        // Compare right-hand side
        search_pattern = pattern->r_exp;
        if (symbol->r_exp->accept(*this) == NULL)
            return NULL;

        return symbol;
    }
};
```

This organization is suitable for the proposed conversion method because the input program, the search patterns, and the generated output are all represented as ASTs. The same tree structure can therefore be traversed several times by different visitors, each one responsible for a specific stage of the transformation. The UML diagram in Figure 4.2 shows the main visitors and helper structures used in this process.

Table 4.1 summarizes the main classes involved in the transformation flow.

Table 4.2 lists the secondary visitors and helper structures used during fragment organization, step resolution, action association, and transition resolution.

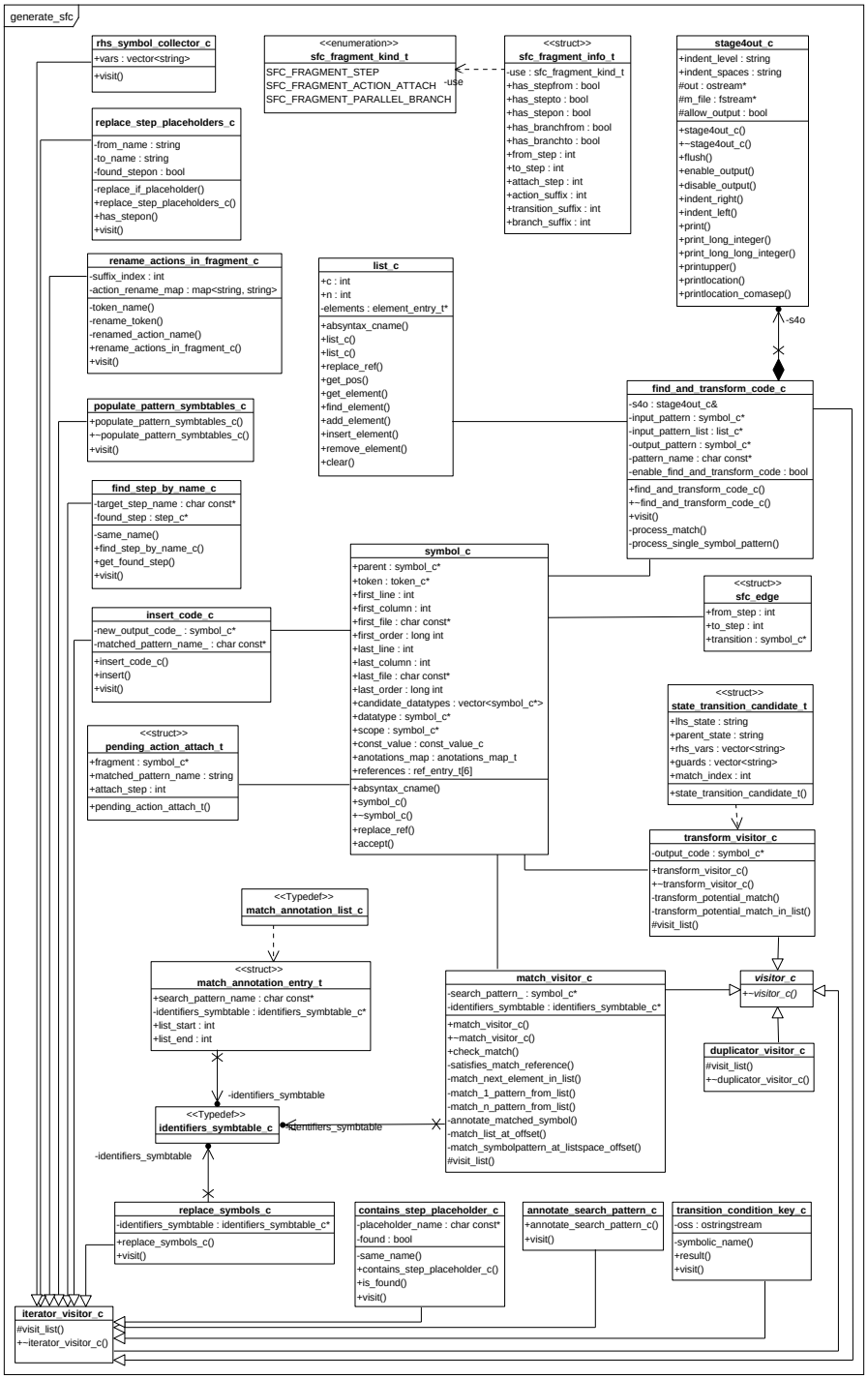


Figure 4.2: UML class diagram of the `generate_sfc.cc` module.

Table 4.1: Main visitors of the `generate_sfc` module and their role in the conversion pipeline.

Class / Pipeline Module	**Purpose / Responsibility**
`populate_pattern_symbtables_c` *Pattern Search and Matching*	Registers the bodies of each `pattern_IN__` and `pattern_OUT__` declaration in the corresponding pattern symbol table, making the pattern catalogue available to the matching stage.
`annotate_search_pattern_c` *Pattern Search and Matching*	Adds the annotations required by the matching algorithm. Symbolic variables are marked as match references, while fixed literals are marked as match tokens.
`match_visitor_c` *Pattern Search and Matching*	Compares subtrees of the input-program AST with the registered input-pattern ASTs. It handles `PATTERN_1__` and `PATTERN_N__`, records identifier mappings, and annotates successful matches in the input-program AST.
`duplicator_visitor_c` *Pattern Search and Matching*	Creates independent copies of AST subtrees. It is used during step resolution, transition aggregation, and action association to duplicate output-pattern templates and generated SFC fragments before they are modified or inserted into the output AST.
`replace_symbols_c` *Pattern Search and Matching*	Replaces placeholder identifiers in copied output patterns with the concrete identifiers captured during pattern search and matching, thereby producing program-specific SFC fragments.
`transform_visitor_c` *Step Resolution; Transition Aggregation; Action Association*	Processes the recorded match annotations and instantiates the corresponding output patterns. It derives state dependencies, assigns concrete steps, prepares transition fragments, and associates action-only fragments with their resolved target steps.
`insert_code_c` *Step Resolution; Transition Aggregation; Action Association*	Inserts the generated fragments into the output AST at the locations represented by pattern placeholders. A `PATTERN_1__` placeholder is replaced directly, whereas fragments associated with `PATTERN_N__` are inserted into the corresponding containing list.
`find_and_transform_code_c` *Step Resolution; Transition Aggregation; Action Association*	Acts as the top-level controller of the module. It selects the body of each Function, Function Block, or Program, invokes pattern matching and transformation, performs the final transition post-processing, and passes the completed SFC AST to the XML-planning and PLCopen XML serialization stages.

Table 4.2: Secondary visitors and helper structures of the generate_sfc module and the main visitors they support.

Class / Supported Main Visitor	**Purpose / Responsibility**
transition_condition_key_c *Supports* find_and_transform_code_c	Builds a deterministic textual key from a transition condition. It is used during final transition post-processing to compare temporary StepExit transitions with real successor transitions having the same source step and condition.
rhs_symbol_collector_c *Supports* transform_visitor_c	Collects the symbolic variables appearing on the right-hand side of an assignment. The collected identifiers are used during step resolution to distinguish state-variable dependencies from guard variables and during action association to identify the target state step.
state_transition_ candidate_t *Supports* transform_visitor_c	Stores an assignment identified as a state-transition candidate, including its left-hand-side state variable, right-hand-side variables, resolved parent state, guard variables, and original match position. These records are used to construct the state-dependency graph and assign concrete SFC steps independently of statement order.
sfc_fragment_info_t *Supports* transform_visitor_c	Stores the result of preparing an instantiated SFC fragment, including its category, detected step placeholders, concrete source and destination steps, action-attachment target, and the suffix values reserved for fragment generation.
sfc_fragment_kind_t *Supports* transform_visitor_c	Enumerates the fragment categories considered by the transformation logic: step-creating fragments, action-attachment fragments, and the parallel-branch category declared in the implementation. In the current flow, only the first two categories are assigned during fragment preparation.
sfc_edge *Supports* transform_visitor_c	Represents an internal edge between two generated SFC steps. The collected edges are currently used to construct an outgoing adjacency map intended to support parallel-path analysis and debugging; the transition field is not yet populated by the transformation flow.
pending_action_attach_t *Supports* transform_visitor_c	Stores an action-only fragment whose target step cannot yet be modified because the corresponding concrete step may not have been inserted into the output network. Pending fragments are processed after state-step generation.
find_step_by_name_c *Supports* transform_visitor_c	Searches an SFC network for a step with a specified name. It is used during action association to locate the temporary StepOn step inside an instantiated fragment and the corresponding concrete step in the generated output network.
replace_step_placeholders_ c *Supports* transform_visitor_c	Replaces temporary step identifiers such as StepFrom, StepTo, and StepOn with concrete generated step names. It also records whether a StepOn placeholder was encountered.

Continued on the next page

Table 4.2 continued from the previous page

Class / Supported Main Visitor	**Purpose / Responsibility**
`contains_step_placeholder_c` *Supports* `transform_visitor_c`	Checks whether an output pattern or instantiated fragment contains a specified temporary step identifier. The result is used to distinguish step-creating patterns from action-attachment patterns and to classify prepared fragments.
`rename_actions_in_fragment_c` *Supports* `transform_visitor_c` *(inactive)*	Renames action definitions and their corresponding action-association references by applying a numerical suffix, thereby preventing name collisions between repeated fragments. The helper is implemented, but its invocation is disabled in the current transformation flow.
`terminal_exit_record_t` *Supports* `find_and_transform_code_c`	Stores an unresolved terminal exit after its temporary `StepExit` transition is removed from the generated SFC AST. Each record contains the source step, fallback target step, and textual exit condition used during final transition post-processing.
`match_annotation_entry_t` *Shared by* `match_visitor_c` *and* `transform_visitor_c`	Stores one successful pattern match, including the matched pattern name, a copy of its identifier-mapping table, and the matched interval when the match occurs inside an AST list. It transfers match information from pattern matching to transformation.
`identifiers_symbtable_c` *Shared by* `match_visitor_c`, `transform_visitor_c`, *and* `replace_symbols_c`	Defines the symbol table used to map pattern placeholders to the concrete identifier tokens found in the input program. The mappings are created during matching, retained in match annotations, and applied when output-pattern copies are instantiated.
`match_annotation_list_c` *Shared by* `match_visitor_c` *and* `transform_visitor_c`	Defines the list of match annotations associated with an AST node. Entries are added by the matching visitor and subsequently ordered and consumed by the transformation visitor when generating SFC fragments.

The entry point of the module is the function `new_code_generator`. This function is called by the MATIEC Stage 4 infrastructure and returns the visitor who will process the annotated program AST.

The pattern-file path is supplied through the Stage 4 option `-O p=<filename>`. The function `stage4_parse_options` extracts this value and stores it in the module-level variable `pattern_file_name`. If no pattern file is provided, `new_code_generator` terminates the conversion with a Stage 4 error.

The pattern file is parsed by calling:

```
stage1_2(pattern_file_name, &pattern_tree_root, false, true).
```

The third argument indicates that the file is not being parsed as a MATIEC library, while the fourth enables the additional pattern-language syntax.

After parsing, the module performs the following sequence:

1. `populate_pattern_symbtables_c` traverses the pattern AST and registers all input and output pattern bodies;
2. `annotate_search_pattern_c` traverses every input pattern and adds the annotations used during matching;
3. the first registered input pattern is selected as the top-level search pattern;
4. a `find_and_transform_code_c` object is created and returned to the MATIEC Stage 4 infrastructure.

The returned visitor processes the body of each Function, Function Block, or Program declaration. For each body, `process_single_symbol_pattern` creates a `match_visitor_c` and attempts to match the top-level pattern against the complete body AST.

When the top-level match succeeds, `process_match` creates an initial output tree from the output pattern associated with the matched top-level pattern. A `transform_visitor_c` then traverses the annotated input AST, instantiates the output patterns associated with the recorded matches, and inserts the resulting SFC fragments into this top-level output tree.

During this transformation stage, each matched output pattern is duplicated, its placeholder identifiers are replaced with the identifiers captured during matching, and the corresponding SFC fragment is prepared for insertion. State fragments are inserted into the generated SFC network, while action-only fragments may be temporarily stored until their target step has been created.

After the generated fragments have been inserted, the module performs the remaining post-processing required to obtain a coherent SFC AST. This includes attaching pending action fragments to the appropriate generated steps and resolving temporary `StepExit` transitions. The resulting AST is then passed to `generate_sfc_xml`.

The execution sequence of the `generate_sfc.cc` module is summarized in Figure 4.3. The diagram shows the main interactions between the Stage 4 entry point, the pattern-processing visitors, the matching and transformation visitors, the helper structures used during fragment generation, and the final XML-generation stage.

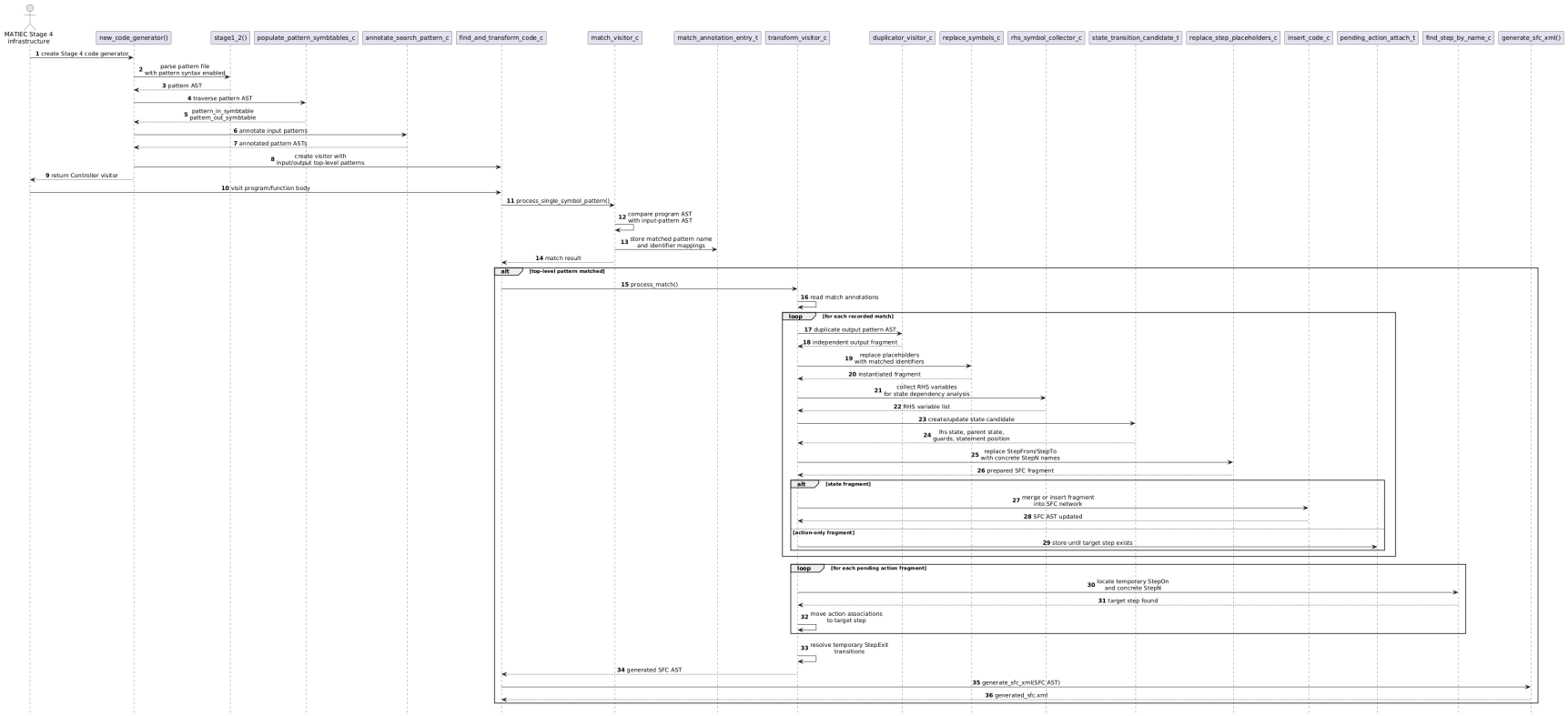


Figure 4.3: Sequence diagram of the `generate_sfc.cc` execution flow.

4.3.2 Pattern Representation and Annotation

The parsed pattern declarations are stored in the two module-level symbol tables shown in Listing 4.2.

Listing 4.2: Symbol tables for input and output patterns.

```
symtable_c<symbol_c *> pattern_in_symbtable;
symtable_c<symbol_c *> pattern_out_symbtable;
```

The keys are the pattern names, and the stored values are pointers to the corresponding AST bodies. The visitor `populate_pattern_symbtables_c` performs this registration. Input patterns are inserted into `pattern_in_symbtable`, while output patterns are inserted into `pattern_out_symbtable`.

Input and output patterns are not grouped in a separate pair object. Their association is established through their common name. During transformation, the name of the matched input pattern is used to retrieve the corresponding output pattern, as shown in Listing 4.3.

Listing 4.3: Retrieval of the output pattern corresponding to the matched input pattern.

```
symbol_c *output_pattern =
    pattern_out_symbtable[matched_pattern_name];
```

If an output pattern with that name is not available, the module reports a Stage 4 error.

The constructs `PATTERN_1__` and `PATTERN_N__` are represented in the pattern AST by nodes of type `match_1_pattern_list_c` and `match_n_pattern_list_c`, respectively. Each node stores the names of the referenced subpatterns.

Before matching begins, the visitor `annotate_search_pattern_c` adds the restrictions required by the matching algorithm. The identifier of each `symbolic_variable_c` is marked with the annotation `"match reference"`, indicating that repeated occurrences must map consistently within one match. Integer literals are marked with `"match token"`, requiring the corresponding source literal to have the same value.

During matching, placeholder bindings are stored in an identifier-mapping table. Since the matcher may test several alternative alignments, especially when processing list patterns or nested pattern references, some bindings are temporary. Before such an attempt begins, the current state of the mapping table is preserved with `push`. If the attempt fails, `pop` restores the previous state, removing the bindings created during the failed attempt and preventing them from affecting subsequent comparisons.

4.3.3 AST Matching and Fragment Resolution

Pattern matching is performed by `match_visitor_c`. Its `check_match` method receives the input-pattern AST, the pattern name, and the program AST subtree to be examined. A new identifier-mapping table is created for the operation before the program subtree accepts the visitor.

The visitor methods are generated for all MATIEC node types through `absyntax.def`. When a program AST node is visited, the current pattern node must have the same exact MATIEC class. The visitor verifies this by casting the pattern node to the expected class. If the cast fails, the nodes represent different language constructs and the comparison fails. If it succeeds, their child references are compared recursively and in their defined order.

For a node with two child references, the generated visitor performs the recursive comparisons illustrated in Listing 4.4.

Listing 4.4: Recursive matching of child nodes against the corresponding pattern subtrees.

```
search_pattern_ = search_pattern_tree->ref1;
if (symbol->ref1 != NULL &&
    symbol->ref1->accept(*this) == NULL)
    return NULL;

search_pattern_ = search_pattern_tree->ref2;
if (symbol->ref2 != NULL &&
    symbol->ref2->accept(*this) == NULL)
    return NULL;
```

A failure in any child comparison causes the complete subtree comparison to fail. String comparisons are performed case-insensitive through `nocase_compare`.

Placeholder bindings are stored in a symbol table whose type is defined in Listing 4.5.

Listing 4.5: Type definition for the identifier symbol table.

```
typedef symtable_c<token_c *> identifiers_symbtable_c;
```

When a placeholder is encountered for the first time during a match, the matcher stores an association between the placeholder used in the pattern and the concrete token found in the program AST. For example, if the pattern variable `Mx` matches the program variable `M1`, this association is stored in the current identifier-mapping table.

If the same placeholder appears again within the same pattern match, it must match with the same program identifier. Therefore, another occurrence of `Mx` must also match `M1`. If it matches a different identifier, the comparison fails. Each occurrence of a pattern is matched with its own mapping table, so the same pattern may match different variables in different rungs of the program.

The current implementation does not perform a separate comparison of the semantic datatype annotations produced by Stage 3. Matching is based on three conditions: the MATIEC node classes must be compatible, fixed tokens must have the required value, and placeholders must remain consistent within the same match.

MATIEC list nodes require a specific comparison strategy because they may contain several statements or several elements of the same symbolic category. The `visit_list` method starts the comparison between a pattern list and a program list. It calls `match_list_at_offset`, which performs the actual element-by-element comparison.

In the simplest case, list matching is strict and ordered. The first pattern element must match the first program element, the second pattern element must match the second program element, and so on. If one element fails to match, the whole list comparison fails. The match only succeeds when all pattern elements have been matched and no unmatched program elements remain.

This strict behavior is more relaxed when the pattern list contains an `any_symbol_c` node. This node represents a variable-length gap in the pattern. It may match zero or more consecutive elements of the program list. When such a node is found, the algorithm recursively tests the remaining pattern elements at different positions of the program list until a valid alignment is found or no more positions are available.

The temporary placeholder bindings created during these recursive attempts are managed through the mapping-table mechanism described in Section 4.3.2. If a tentative alignment fails, the bindings created during that attempt are discarded before the matcher tries the next possible alignment.

The nodes `match_1_pattern_list_c` and `match_n_pattern_list_c` are used to match references to other patterns. When the visitor encounters a `match_1_pattern_list_c` node, `match_1_pattern_from_list` tests the referenced patterns in their declared order against the same program subtree. The first successful pattern is recorded, and failed attempts do not affect the mappings used by later attempts.

For a `match_n_pattern_list_c` node, `match_n_pattern_from_list` searches the program list for multiple occurrences of the referenced patterns. At each list position, the referenced patterns are tested in their declared order. If one pattern matches, that occurrence is annotated and the search continues. If none of the patterns matches at the current position, the current program element is skipped and the scan proceeds to the next one.

For each successful occurrence, `annotate_matched_symbol` records:

- the name of the matched pattern;
- an independent copy of the identifier mappings;
- the first and last matched list indexes, when the match occurs inside a list.

The information is stored in the annotation map of the matched program AST node under the key shown in Listing 4.6.

Listing 4.6: Annotation key used to store successful pattern matches.

```
generate_sfc: matched to pattern
```

The function also prints a diagnostic message that indicates the successful pattern match. However, at this stage, the conversion has not yet produced the final SFC structure. The matching phase only records which program subtrees correspond to known input patterns and stores the identifier mappings required to instantiate the associated output patterns.

The annotations produced by `match_visitor_c` are subsequently consumed by `transform_visitor_c`. This second phase interprets the matched assignments in the context of the complete

statement list, resolves the dependencies between detected state variables, replaces temporary step placeholders, associates action-only fragments with the appropriate generated steps, and removes or records temporary exit transitions. In this way, the local pattern matches are transformed into a coherent SFC AST.

The transformation of list-based matches is performed in two passes. This is necessary because an action assignment may appear before or after the state equation that creates its target step.

During the first pass, `transform_potential_match_in_list` processes only patterns whose output contains `StepFrom` and `StepTo`. For each matched assignment, `collect_transition_candidate` records:

- the variable on the left-hand side of the assignment;
- all symbolic variables appearing on its right-hand side;
- the position of the matched statement in the original statement list.

These data are stored in a `state_transition_candidate_t` structure. After all candidates have been collected, `resolve_state_candidate_parents` determines their state dependencies.

The set of variables assigned on the left-hand side is first collected. For each LHS state candidate, the first right-hand-side variable that is also a detected state variable, excluding the candidate's own variable, is selected as its parent state. The remaining right-hand-side variables are stored as guard variables.

For example, in:

```
M1 := NOT X2 AND (M1 OR (M0 AND X1))
```

`M1` is the state being created, `M0` is identified as its parent state, and `X1` and `X2` remain part of the transition conditions defined by the matched output pattern.

The function `assign_steps_by_state_dependencies` then assigns one SFC step number to each detected state variable. States without a detected parent are treated as roots and are connected from `Step0`. Child states are visited after their parent through a depth-first traversal of the dependency graph.

The resulting association between state variables and SFC step numbers is stored in the map shown in Listing 4.7.

Listing 4.7: Mapping from state variables to generated SFC step numbers.

```
std::map<std::string, int> state_var_to_step;
```

For example:

$$\texttt{M0} \mapsto \texttt{Step1}, \qquad \texttt{M1} \mapsto \texttt{Step2}.$$

The generated SFC is constructed by considering both the dependencies between state variables and the execution order inherited from the original LD rungs. State dependencies determine the predecessor–successor relations between steps. The original statement positions determine the order in which matched fragments are processed, while step numbers are assigned through a deterministic depth-first traversal of the dependency graph.

During the second pass, each state fragment receives its concrete source and destination step numbers. If the candidate has a recognized parent state, the source step is obtained from the parent's entry in `state_var_to_step`. Otherwise, the source is `Step0`. The destination step is the step assigned to the state on the left-hand side.

Action-only patterns are identified by an output fragment containing `StepOn` and not `StepFrom` or `StepTo`. The function `find_attach_step_from_rhs` examines the right-hand side of the matched assignment and searches for a variable already present in `state_var_to_step`.

For example, for:

```
Y0 := M1
```

the variable `M1` identifies the step to which the action `Y0` must be associated.

If the target step has not yet been inserted into the output network, the fragment is stored in a `pending_action_attach_t` structure. After all state fragments have been generated, `move_stepon_action_associations_to_step`:

1. locates the temporary `StepOn` node in the action fragment;
2. locates the concrete `StepN` node in the main output;
3. copies the action associations from `StepOn` to `StepN`;
4. removes the temporary `StepOn` node.

A transition to `StepExit` represents the deactivation condition of a state before its final destination is known. After all state fragments have been generated, the tool checks whether the same source step already has a transition with the same condition leading to a real successor step.

When such a transition exists, it already represents the state change, so the temporary transition to `StepExit` is removed. When no real successor transition exists, the source step and condition are stored in `terminal_exit_records`, and the temporary StepExit transition is removed from the generated SFC AST. These records are retained for diagnostic and post-processing purposes, and are not currently passed to the XML exporter.

The generated state relations determine the transitions inserted into the final SFC AST. The `generate_sfc_xml` module subsequently traverses this AST and reconstructs its own incoming and outgoing relations to plan the required PLCopen XML divergences, convergences, jumps, and graphical connections.

4.3.4 Output Pattern and SFC Fragment Organization

After a program subtree has been matched and annotated, the corresponding SFC structure is generated by `transform_visitor_c`. For each recorded match, the visitor retrieves the output-pattern AST from `pattern_out_symbtable` using the name of the matched input pattern.

The registered output pattern acts as a template and is not modified directly. The selected output-pattern subtree is first duplicated through `duplicator_visitor_c`, as shown in Listing 4.8.

Listing 4.8: Duplication of the output-pattern AST before fragment instantiation.

```
symbol_c *new_output_code =
    (symbol_c *)output_pattern->accept(duplicator_visitor);
```

The duplicated fragment is independent from the registered pattern template. List nodes are recreated and populated with copies of their elements, token nodes are recreated with duplicated values, and nodes with child references are reconstructed recursively.

The placeholder bindings recorded during matching are then applied by `replace_symbols_c`. This visitor traverses the copied fragment and replaces each mapped `identifier_c` value with the corresponding source identifier, as shown in Listing 4.9.

Listing 4.9: Replacement of pattern placeholders with captured source identifiers.

```
token_c *new_token =
    dynamic_cast<token_c *>(
        (*identifiers_symbtable)[symbol->value]);

if (new_token != NULL)
    symbol->value = strdup(new_token->value);
```

A pattern placeholder such as `Mx` may therefore be replaced by a concrete identifier such as `M1`. However, a single placeholder does not currently capture an entire Boolean expression such as `X1 AND SafetyOK`. To support this type of condition, the corresponding operation node and its operands must be explicitly represented in both the input and output pattern ASTs.

The generated fragment may contain MATIEC SFC nodes such as:

- `sfc_network_c`, containing the generated SFC elements;
- `initial_step_c`, representing the initial step;
- `step_c`, representing a regular SFC step;
- `transition_c`, representing a transition between steps;
- `action_association_c`, representing an action associated with a step.

Output patterns use temporary step names because the final step identifiers are only known after the state dependencies have been analyzed. The currently used placeholders are:

- StepFrom, representing the source step of a generated transition;
- StepTo, representing the new step created by a state pattern;
- StepOn, representing a temporary step used by an action-only pattern;
- StepExit, representing a temporary destination for a state-deactivation transition.

In this context, an *action-only fragment* is a generated SFC fragment that does not create a new process state. Instead, it represents an assignment that should be associated as an action of an already generated SFC step. This occurs when an ST assignment does not contain a self-holding state equation, but its right-hand side refers directly to a state variable already detected by another pattern.

For example, the assignment

```
Y0 := M1
```

does not define a new retained state. The variable Y0 is active while the state M1 is active. Therefore, instead of generating a new SFC step for Y0, the corresponding output pattern creates a temporary step named StepOn, as shown in Listing 4.10.

Listing 4.10: Temporary SFC step used to attach a Boolean action to a resolved state step.

```
STEP StepOn:
  Y0(N);
END_STEP
```

This temporary step is not part of the final SFC structure. It is used only as an intermediate container for the action association. After the concrete step associated with M1 has been identified, the action association Y0(N) is moved from StepOn to that step, and the temporary StepOn node is removed.

By contrast, a state fragment contains StepFrom and StepTo, because it creates a new step and the transitions required to enter and leave that step.

The function prepare_sfc_fragment examines the copied output pattern and classifies the resulting fragment according to the placeholders it contains. A fragment containing StepFrom and StepTo is treated as a state fragment, while a fragment containing only StepOn is treated as an action-attachment fragment.

For state fragments, replace_step_placeholders_c replaces StepFrom and StepTo with concrete names such as Step1 and Step2. The concrete step numbers are obtained from the state-dependency analysis described in Section 4.3.3. Action-only fragments retain the temporary StepOn step until the corresponding real step has been inserted into the output network, as described in the action-association stage.

Each instantiated fragment is also annotated with the name of the pattern from which it was generated, as shown in Listing 4.11.

Listing 4.11: Annotation recording the pattern used to generate an SFC fragment.

```
new_output_code->anotations_map[
    "generate_sfc: generated from pattern name"
] = (symbol_c *)matched_pattern_name;
```

This annotation is used for traceability and debugging.

Output patterns may themselves contain references to other patterns. For example, the output of a top-level pattern may contain a `PATTERN_N__` node indicating where the fragments generated from its subpatterns must be inserted. After copying the top-level output pattern, `transform_visitor_c` temporarily uses this copy as the current output tree and processes the matches found inside the corresponding input structure. The generated subpattern fragments are inserted into the copied top-level output before the completed structure is inserted into the main output AST.

The generic insertion operation is performed by `insert_code_c`. For a `match_1_pattern_list_c` placeholder, the parent reference is redirected to the generated fragment through the operation shown in Listing 4.12.

Listing 4.12: Replacement of a single-match placeholder with the generated fragment.

```
symbol->parent->replace_ref(symbol, new_output_code_);
```

For a `match_n_pattern_list_c` placeholder, the parent must be a `list_c`. The generated fragment is inserted into that list without removing the placeholder, allowing further matched occurrences to be added at the same location.

The top-level output pattern creates the main `sfc_network_c` node that contains the generated chart. Since each local fragment may also contain its own `sfc_network_c`, the fragment network is not inserted as a nested network. Instead, `merge_sfc_fragment_into_output` copies the steps, transitions, and actions contained in the local fragment network and appends them directly to the main SFC network. The resulting structure is the SFC AST that is later passed to `generate_sfc_xml`.

4.4 `generate_sfc_xml` Module

The `generate_sfc_xml.cc` module converts the SFC AST produced by `generate_sfc` into a PLCopen XML project. The generated document can follow either the Beremiz or CODESYS format, selected through the module-level `XML_DIALECT` constant. The current configuration targets CODESYS.

The implementation separates analysis from serialization. An initial pass collects the information required to represent the SFC in XML, while a subsequent pass writes the project structure and its SFC elements. The classes and operations involved in these stages are described in the following subsections.

4.4.1 Module Organisation and Execution Flow

The generate_sfc_xml.cc module converts the SFC AST produced by generate_sfc.cc into a PLCopen XML project. The implementation is organized as a set of visitors with separate responsibilities for variable collection, action collection, graph analysis, identifier assignment, ST expression serialization, and XML output.

The UML diagram in Figure 4.4 shows the main visitors and helper structures used in this process.

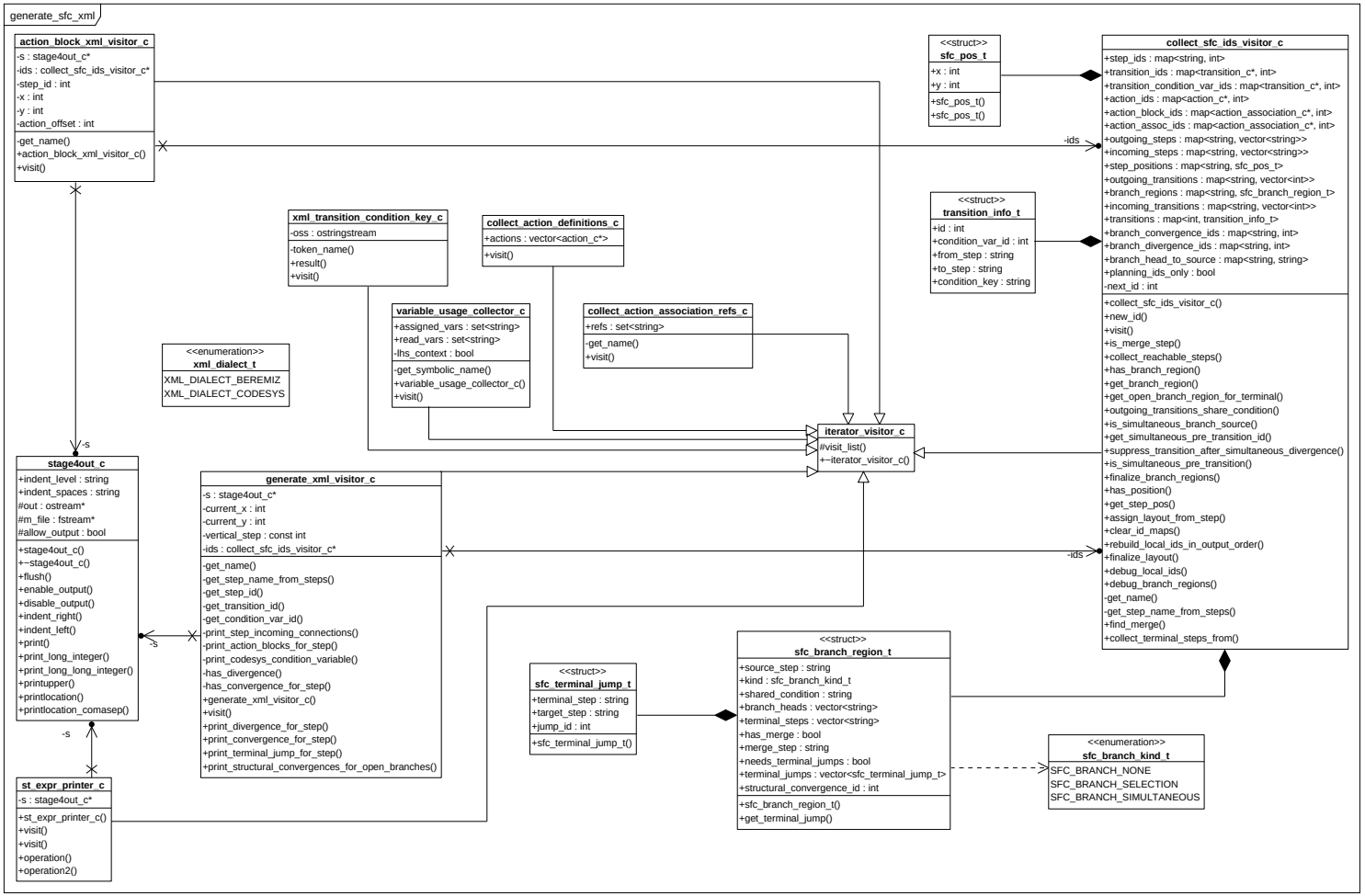


Figure 4.4: UML class diagram of the `generate_sfc_xml.cc` module.

Table 4.3 summarizes the main classes involved in the XML generation flow.

Table 4.3: Main classes of the generate_sfc_xml module.

Class	Purpose / Responsibility
st_expr_printer_c	Serialises supported ST expressions and statements into textual form. It is used when writing transition conditions and ST action bodies to the XML document. The visitor handles symbolic variables, Boolean literals, logical expressions, assignments, and statement lists.
variable_usage_collector_c	Collects the symbolic variables read and assigned in the generated AST. Variables that are read but not assigned are classified as input variables, while assigned variables are classified as local variables.
xml_transition_condition_key_c	Produces a normalised textual representation of a transition condition. These keys are used to compare the outgoing conditions of a step and determine whether the resulting branch is a selection divergence or a simultaneous divergence.
collect_sfc_ids_visitor_c	Performs the main analysis pass over the SFC AST. It assigns local identifiers to steps, transitions, actions, and action blocks; records incoming and outgoing graph relations; detects divergence and convergence regions; plans terminal jumps; and calculates the graphical position of each step.
action_block_xml_visitor_c	Generates the XML representation of the action associations attached to a step. It assigns their previously collected identifiers, positions each action block relative to its step, and writes the dialect-specific action representation.
collect_action_definitions_c	Collects the action_c nodes that define explicit SFC actions. The resulting list is used when writing the POU action definitions and when distinguishing real actions from direct Boolean action variables.
collect_action_association_refs_c	Collects the names referenced by SFC action associations. References that do not correspond to explicit action definitions are added to the POU interface as Boolean variables when generating CODESYS XML.
generate_xml_visitor_c	Performs the final serialization pass. It writes the XML elements for initial steps, regular steps, transitions, action blocks, divergences, convergences, and terminal jump steps using the identifiers, branch information, and positions prepared by collect_sfc_ids_visitor_c.

The module also defines several helper structures used during XML planning. The sfc_pos_t structure stores the graphical coordinates of a step, while transition_info_t records the source, destination, identifier, condition-variable identifier, and normalised condition of a transition.

Branch information is stored in `sfc_branch_region_t`, including the source step, branch type, branch heads, detected merge step, terminal steps, and any terminal jumps required to close an open branch. Each terminal jump is represented by `sfc_terminal_jump_t`. The identifiers of divergences and natural convergences are maintained separately by `collect_sfc_ids_visitor_c`.

Listing 4.13 presents the function signature of the module's entry point.

Listing 4.13: Function signature of the PLCopen XML generation entry point.

```
void generate_sfc_xml(symbol_c *tree_root,
                      stage4out_c *s4o)
```

The parameter `tree_root` refers to the generated SFC AST, while `s4o` is the output stream used to write the PLCopen XML document. In the current pipeline, this stream is created by `generate_sfc.cc` using `stage4out_c` and is configured to write the file `generated_sfc.xml`.

The XML generation process is applied at body level. The analysis described in this subsection is repeated for each generated Function, Function Block, or Program body found in the AST. In the current conversion flow, this normally corresponds to the generated Program body. For each body, the function first traverses the corresponding generated SFC AST with `variable_usage_collector_c`. This pass collects the variables that are read and assigned in that body. Variables that are assigned are classified as local variables, while variables that are only read are classified as input variables.

The next stage creates a `collect_sfc_ids_visitor_c` object and traverses the SFC AST to collect the information required for XML planning. This pass assigns preliminary identifiers, records transitions and step connections, collects incoming and outgoing relations, and stores transition information. During this process, `xml_transition_condition_key_c` is used to obtain comparable textual keys for transition conditions.

After this first analysis pass, `collect_sfc_ids_visitor_c` finalizes the branch regions, rebuilds the local identifiers in output order, and calculates the graphical layout of the SFC elements. This preparation is required because PLCopen XML represents steps, transitions, action blocks, divergences, convergences, and jumps as separate graphical objects connected through numerical references.

For the CODESYS dialect, the module also collects action-association references and explicit action definitions. The visitor `collect_action_association_refs_c` collects the names used in step action associations, while `collect_action_definitions_c` collects the actual `action_c` definitions present in the AST. Action-association names that do not correspond to explicit action definitions are treated as direct Boolean action variables and are added to the generated POU interface when required.

After the interface and optional action definitions have been written, the SFC body is serialized. A `generate_xml_visitor_c` object traverses the SFC AST and writes the XML representation of the SFC elements using the identifiers, branch regions, terminal jumps, and positions prepared by `collect_sfc_ids_visitor_c`. When a step contains action associations,

generate_xml_visitor_c delegates their serialization to action_block_xml_visitor_c. Transition conditions and ST action bodies are printed through st_expr_printer_c.

The generated document contains a Program POU with its interface, optional action definitions, and SFC body, followed by the configuration, resource, task, and POU instance required by the selected XML dialect. The current implementation does not yet validate the generated document against the PLCopen XML schema.

The execution sequence of the generate_sfc_xml.cc module is summarized in Figure 4.5. The diagram shows the main interactions between the XML generation entry point, the analysis visitors, the action/interface collectors, the final XML visitor, and the output stream.

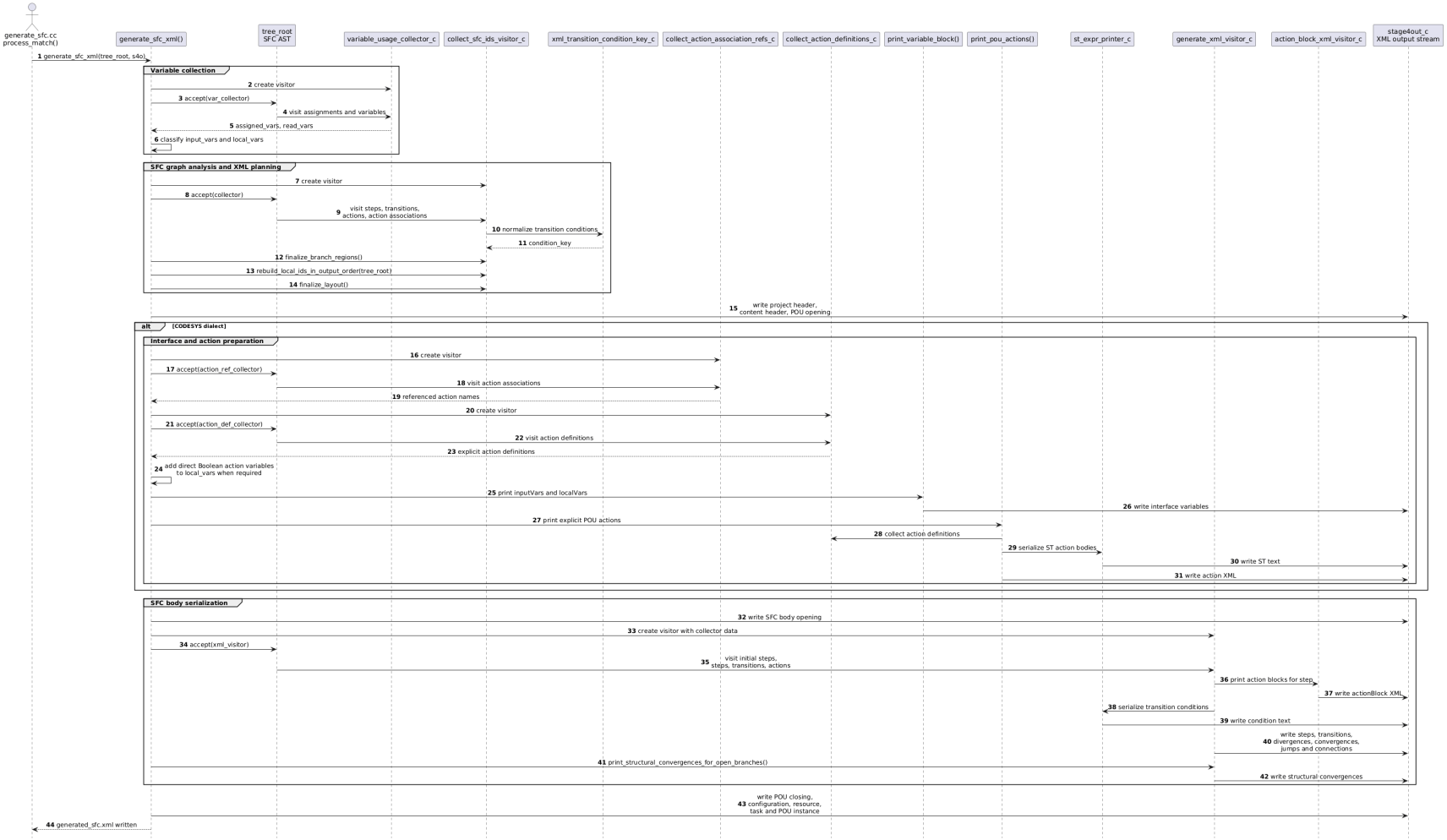


Figure 4.5: Sequence diagram of the `generate_sfc_xml.cc` execution flow.

4.4.2 SFC AST Analysis and XML Planning

The generated SFC received as input cannot be serialized directly by converting each AST node independently. The MATIEC AST represents the symbolic structure of the SFC, whereas PLCopen XML represents steps, transitions, action blocks, and branch elements as separate graphical objects connected through numerical references. The module must therefore analyze the complete SFC AST before writing the XML document.

For example, an AST transition connecting `Step1` to `Step2` contains references to both step names. In PLCopen XML, the two steps and the transition are written as separate elements. Their relationship is expressed through the `localId` attribute assigned to each element and the corresponding `refLocalId` values used by their connection points. Consequently, the exporter must first determine all elements and connections that will be required in the XML representation.

The preparation stage begins by collecting the declarations required by the generated Program POU. The visitor `variable_usage_collector_c` distinguishes the variables according to their use in the generated AST. A variable that is read, but is never assigned, is classified as an input, whereas an assigned variable is classified as local.

Action-association names are compared with the actual SFC actions present in the AST. If a name corresponds to an `action_c` node, it is treated as a reference to that action definition. Otherwise, it is treated as a Boolean action variable and, for current CODESYS output, added to the local variables of the generated POU unless it has already been declared as an input.

For example, Listing 4.14 shows a textual SFC association of `OpenValve` with `Step1` using the `N` qualifier.

Listing 4.14: Association of the OpenValve action with Step1 using the N qualifier.

```
STEP Step1:
    OpenValve(N);
END_STEP
```

For example, Listing 4.14 shows a textual SFC association of `OpenValve` with `Step1` using the `N` qualifier.

Listing 4.15: Definition of the OpenValve action with an ST body.

```
ACTION OpenValve:
    ValveOutput := TRUE;
    Angle := TargetAngle;
END_ACTION
```

If the action definition shown in Listing 4.15 is also present in the AST, `OpenValve` is linked to this action, whose statements are executed while the action is active. If no `ACTION OpenValve` definition exists, `OpenValve` is instead treated as a Boolean variable activated by `Step1`. Both assignments in the action definition are preserved in the ST body of the corresponding action element in the generated XML. For CODESYS output, the implementation collects variables from the generated AST and declares assigned variables as local variables and variables

that are only read as inputs. However, these declarations currently use `BOOL`, without preserving the original variable types.

The structural analysis is performed by `collect_sfc_ids_visitor_c`. During its first traversal, the visitor records every step and transition present in the SFC AST. For each transition, it stores:

- the source and destination step names;
- the transition condition;
- the incoming and outgoing relations of the connected steps;
- temporary identifiers for the transition and any associated XML elements.

The incoming and outgoing relations form an internal directed graph of the generated SFC. This graph is used to introduce divergences, convergences, and terminal jumps that are not explicitly represented in the SFC AST but are required to produce a structurally valid PLCopen XML document that can be imported by the target development environment.

A branch region is detected when a step has more than one successor. The conditions of its outgoing transitions are converted into textual keys by `xml_transition_condition_key_c`. These keys preserve the structure of the supported Boolean expressions and allow the exporter to distinguish two branch forms:

$$\text{selection:} \quad \text{source step} \rightarrow \text{divergence} \rightarrow \text{individual transitions} \rightarrow \text{branch steps},$$

$$\text{simultaneous:} \quad \text{source step} \rightarrow \text{shared transition} \rightarrow \text{divergence} \rightarrow \text{branch steps}.$$

When the outgoing transitions have the same non-empty condition key, the region is classified as simultaneous. The first outgoing transition is then retained as the shared transition preceding the divergence, while the remaining equivalent transitions are omitted from the XML plan. Their destination steps are connected directly to the simultaneous divergence.

When the condition keys differ, the region is classified as a selection branch. In this case, each branch retains its own transition and condition. After identifying a divergence, the module searches for a step that can be reached from all its branches. The reachable steps of each branch are collected, and the first common step encountered by the implementation is selected as the convergence merge step. Then a convergence XML element is planned immediately before this step.

For example, Listing 4.16 represents two outgoing transitions from `Step2` with the same condition.

Listing 4.16: Outgoing transitions with identical conditions used by the exporter to identify a simultaneous divergence.

```
TRANSITION FROM Step2 TO Step3
```

```
  := X2;

TRANSITION FROM Step2 TO Step4
  := X2;
```

Both outgoing transitions have the same non-empty condition key, `X2`. The region is therefore classified as a simultaneous divergence. In the XML plan, only one transition with condition `X2` is kept before the divergence, and the two destination steps, `Step3` and `Step4`, are connected as simultaneous branches.

By contrast, Listing 4.17 represents two outgoing transitions from `Step2` with different conditions.

Listing 4.17: Outgoing transitions with different conditions used by the exporter to identify a selection divergence.

```
TRANSITION FROM Step2 TO Step3
  := X2;

TRANSITION FROM Step2 TO Step4
  := X3;
```

the outgoing condition keys are different. The region is therefore classified as a selection divergence. In this case, each branch keeps its own transition and condition: one branch is enabled by `X2`, while the other is enabled by `X3`.

After the divergence has been classified, the module analyzes the steps reachable from each branch. If, for example, both branches can later reach `Step5`, that step is selected as the merge point and a convergence element is planned immediately before it in the XML representation.

If the branches have no common reachable step, the branch region is considered open. The terminal step of each branch is associated with a `jumpStep` targeting `Step0`. For CODESYS output, an additional structural convergence is generated only to allow the open-branched region to be imported correctly. This convergence does not represent a semantic merge in the original SFC, since each terminal path already returns to `Step0` through its corresponding jump.

Once all branch regions and additional XML objects are known, the temporary identifiers assigned during the first traversal are discarded. `rebuild_local_ids_in_output_order` performs a second traversal and assigns the final `localId` values.

This second allocation pass is necessary because the complete set of XML objects is only known after the graph has been analyzed. In addition to the steps and transitions already present in the AST, identifiers may be required for:

- divergence and convergence elements;
- action blocks and action associations;
- terminal jump steps;

- CODESYS transition-condition variables.

The resulting identifiers provide the reference system used during serialisation. Each object receives a unique `localId`, and the connections between objects are later expressed through `refLocalId` attributes.

Graphical coordinates are assigned only after the structural XML plan has been completed. The layout starts at `Step0`; single successors are placed vertically below their predecessors, while multiple successors are distributed horizontally. These coordinates determine how the generated SFC is displayed after import, but do not alter its logical topology or its identifier-based connections.

4.4.3 PLCopen XML Serialization

The final pass is performed by `generate_xml_visitor_c`. The visitor traverses the generated SFC AST and writes the corresponding PLCopen XML elements using the identifiers and structural information prepared during the analysis stage.

Both `initial_step_c` and `step_c` nodes are serialised as `step` elements. The initial step is distinguished through the `initialStep="true"` attribute. Each regular step contains an input connection that refers to its preceding transition or convergence through `refLocalId`.

A `transition_c` node is serialised as a `transition` element. Its input connection refers to the preceding step or divergence. The destination step is subsequently connected to the transition identifier.

Transition conditions are represented differently in the two supported formats. For CODESYS output, the transition condition is written as a separate `inVariable` element connected to the transition. For Beremiz output, the Boolean expression appearing after `:=` in the textual SFC transition is written directly inside the transition as inline ST. In both cases, `st_expr_printer_c` serialises this expression from the `transition_condition_c` subtree.

Action associations are serialised by `action_block_xml_visitor_c`. Each association produces an `actionBlock` connected to its owning step. For CODESYS, the block contains an action reference with the corresponding qualifier. For Beremiz, the referenced action name is written as inline ST.

Explicit `action_c` nodes are written separately in the CODESYS POU action section. Their statement bodies are serialised as ST, while the corresponding action blocks refer to the action definitions by name.

The divergences, convergences, and terminal jumps prepared during the analysis stage are inserted at their planned positions in the SFC body. All generated objects are written as separate XML elements, with their relationships expressed through `localId` and `refLocalId` references.

Chapter 5

Test and Validation

This chapter presents the results obtained with the implemented LD-to-SFC conversion tool. The evaluation comprises proof-of-concept test cases and a larger example adapted from the literature. The generated SFC structures and PLCopen XML projects are analyzed, together with their import and graphical representation in CODESYS.

5.1 Evaluation Method

The proof-of-concept test cases were designed to isolate the main conversion capabilities of the tool, including state recognition, sequential transitions, action associations, divergences, convergences, and open branch regions. The literature example combines several of these structures in a single program and is used to assess the behavior of the complete conversion process.

For each test case, the original LD program was converted into ST using Beremiz. The resulting ST program was processed by the modified MATIEC compiler together with the external pattern file. The generated SFC AST was then serialized as a PLCopen XML project and imported into CODESYS.

The result of each conversion was evaluated by comparing the generated SFC with the expected state dependencies, transition conditions, actions, and branch topology of the original LD program. The generated XML connections were also examined, and successful import into CODESYS was used to confirm that the graphical representation was accepted and displayed correctly.Any differences between the expected and generated results were identified in the discussion of the corresponding test case.

The tests were executed on a development computer running Windows 11. The modified MATIEC compiler was built and executed in an Ubuntu-22.04 environment through Windows Subsystem for Linux. Beremiz was used to prepare the input LD programs and export their corresponding ST representations, while CODESYS was used to import and visually inspect the generated PLCopen XML projects.

The same MATIEC implementation and conversion-pattern catalog were used throughout the evaluation, ensuring that the results were obtained under consistent execution conditions.

The current implementation does not yet validate the generated document against the PLCopen XML schema. Consequently, successful import into CODESYS is used as a practical compatibility check, but is not treated as proof of complete conformance with every requirement of the PLCopen XML specification.

5.2 Test Cases and Results

This section presents the test programs used to exercise the main conversion structures supported by the implemented tool. For each test, the input LD program, its ST representation, and the SFC obtained after import into CODESYS are presented. A combined discussion of the results is provided in Section 5.4.

5.2.1 Test 1: Single State and Associated Output

The first test considers a single feedback state and an output controlled by that state. The input LD program is shown in Figure 5.1.

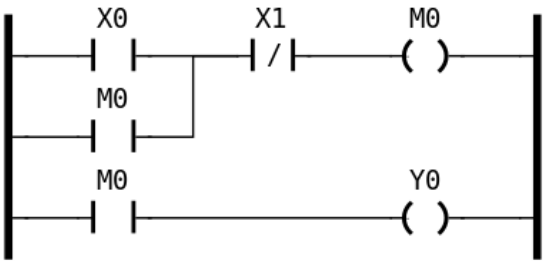


Figure 5.1: Input LD program for Test 1.

The first rung defines the state variable `M0`. The input `X0` activates the state, the parallel `M0` contact maintains it, and the normally closed `X1` contact defines its deactivation condition. The second rung associates the output `Y0` with the active state `M0`.

The ST program generated from this LD representation is:

```
M0 := NOT X1 AND (M0 OR X0);
Y0 := M0;
```

The state assignment matches pattern `p3`, while the assignment `Y0 := M0` matches pattern `p6`. Both patterns are defined in the conversion catalog presented in Appendix A. Together, they generate the state step, its entry and exit transitions, and the output associated with the detected state.

The SFC generated by the tool and imported into CODESYS is shown in Figure 5.2.

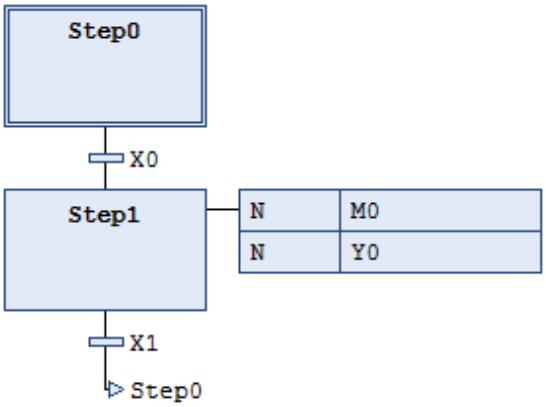


Figure 5.2: Generated SFC for Test 1 after import into CODESYS.

The SFC contains the initial step `Step0`, followed by a transition with condition `X0` and the state step `Step1`. The actions `M0` and `Y0` are associated with `Step1`. A transition with condition `X1` leaves `Step1` and returns control to `Step0`.

5.2.2 Test 2: Sequential State Chain

The second test considers four feedback states connected through a sequential dependency chain. The input LD program is shown in Figure 5.3.

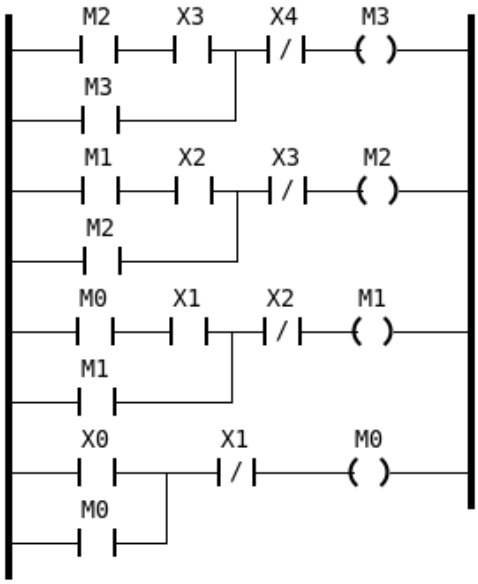


Figure 5.3: Input LD program for Test 2.

The program defines the state variables `M0`, `M1`, `M2`, and `M3`. The input `X0` activates `M0`. Each subsequent state is activated by the preceding state together with its corresponding transition input. The final input `X4` deactivates `M3` and terminates the sequence.

The ST program generated from the LD representation is:

```
M3 := NOT X4 AND (M3 OR (M2 AND X3));
M2 := NOT X3 AND (M2 OR (M1 AND X2));
M1 := NOT X2 AND (M1 OR (M0 AND X1));
M0 := NOT X1 AND (M0 OR X0);
```

Each assignment matches the sequential-state pattern defined in the conversion catalogue presented in Appendix A. From the references between the state variables, the tool determines the following dependency chain:

$$\texttt{M0} \rightarrow \texttt{M1} \rightarrow \texttt{M2} \rightarrow \texttt{M3}.$$

The SFC generated by the tool and imported into CODESYS is shown in Figure 5.4.

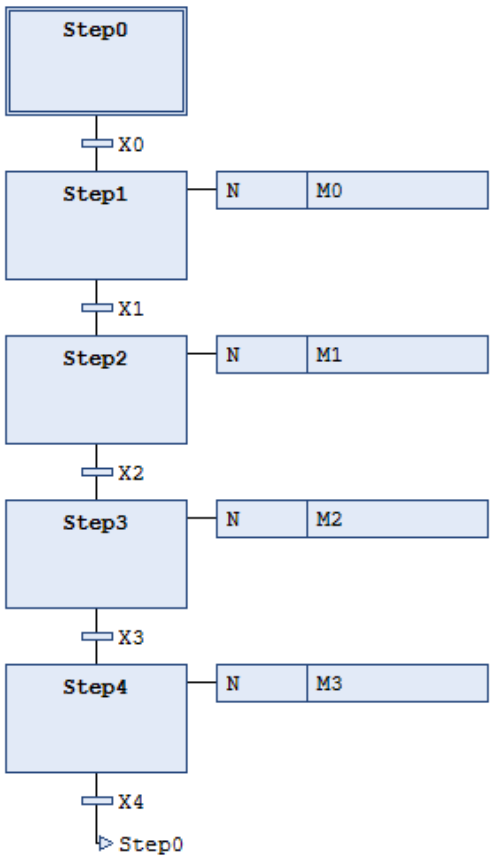


Figure 5.4: Generated SFC for Test 2 after import into CODESYS.

The SFC begins at `Step0`. The condition `X0` leads to `Step1`, which contains the action `M0`. The conditions `X1`, `X2`, and `X3` subsequently connect the steps associated with `M1`, `M2`, and `M3`, respectively. The final transition, conditioned by `X4`, returns the execution from `Step4` to `Step0`.

5.2.3 Test 3: Simultaneous Divergence

The third test considers a state sequence containing two successor states activated from the same preceding state. One branch terminates directly, while the other continues to an additional successor state. The input LD program is shown in Figure 5.5.

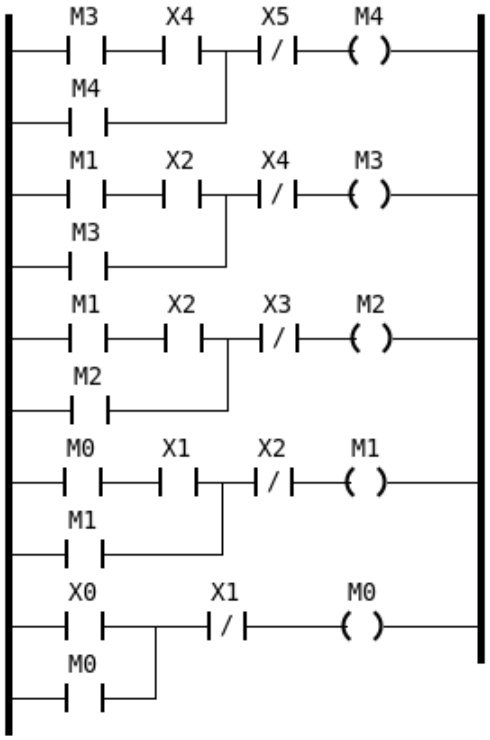


Figure 5.5: Input LD program for Test 3.

The program defines the state variables `M0`, `M1`, `M2`, `M3`, and `M4`. The states `M0` and `M1` form the initial sequential part of the program. From `M1`, both `M2` and `M3` are activated by `X2`. The state `M2` is deactivated by `X3`, whereas `M3` is followed by `M4` when `X4` becomes active. The final state `M4` is deactivated by `X5`.

The relevant ST assignments generated from the LD representation are:

```
M4 := NOT X5 AND (M4 OR (M3 AND X4));
M3 := NOT X4 AND (M3 OR (M1 AND X2));
M2 := NOT X3 AND (M2 OR (M1 AND X2));
M1 := NOT X2 AND (M1 OR (M0 AND X1));
M0 := NOT X1 AND (M0 OR X0);
```

These assignments match the state patterns defined in the conversion catalog presented in Appendix A. The dependencies identified between the state variables are:

$$\texttt{M0} \rightarrow \texttt{M1} \rightarrow \begin{cases} \texttt{M2}, \\ \texttt{M3} \rightarrow \texttt{M4}. \end{cases}$$

Since the transitions from `M1` to `M2` and `M3` share the same condition, `X2`, they are represented by a common transition followed by a simultaneous divergence.

The SFC generated by the tool and imported into CODESYS is shown in Figure 5.6.

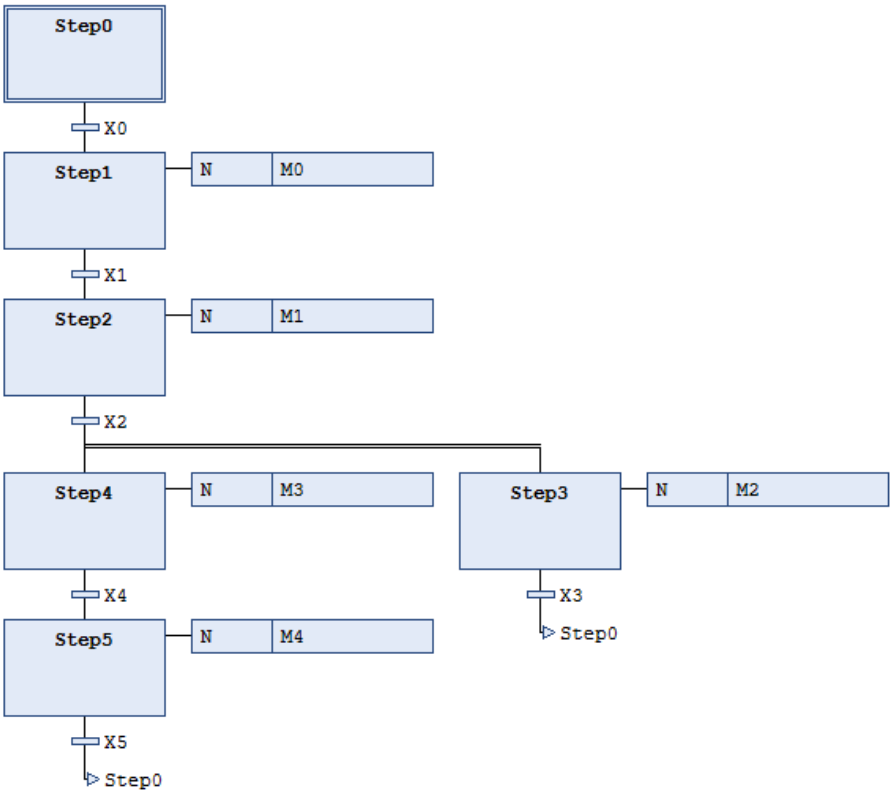


Figure 5.6: Generated SFC for Test 3 after import into CODESYS.

The SFC starts with the transitions `Step0` $\xrightarrow{\texttt{X0}}$ `Step1` and `Step1` $\xrightarrow{\texttt{X1}}$ `Step2`, representing the states `M0` and `M1`. The condition `X2` then precedes a simultaneous divergence leading to `Step3`, associated with `M2`, and `Step4`, associated with `M3`.

On the first branch, the condition `X3` returns the execution from `Step3` to `Step0`. On the second branch, `X4` connects `Step4` to `Step5`, which represents `M4`, and `X5` subsequently returns this branch to `Step0`.

5.2.4 Test 4: Selection Divergence

The fourth test considers a branching structure in which two alternative successor states can be reached from the same preceding state. Unlike the previous test, the two outgoing branches are activated by different transition conditions, representing a selection divergence rather than a simultaneous divergence. The input LD program is shown in Figure 5.7.

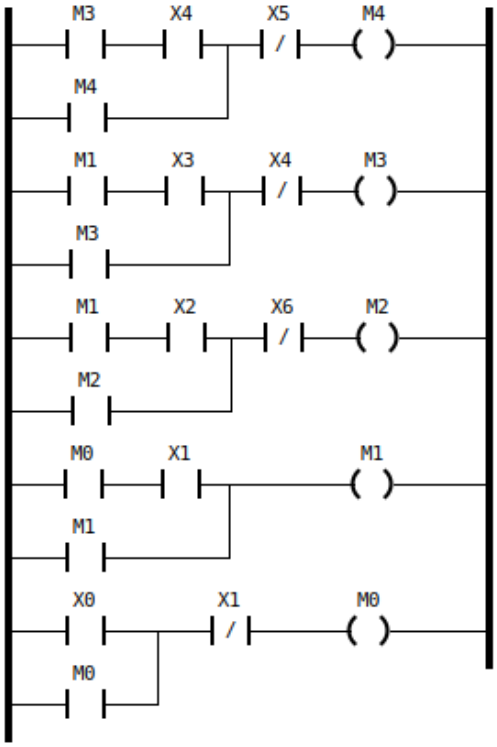


Figure 5.7: Input LD program for Test 4.

The program defines the state variables `M0`, `M1`, `M2`, `M3`, and `M4`. The states `M0` and `M1` form the initial part of the sequence. From `M1`, two alternative successor states can be activated: `M2` through condition `X2`, and `M3` through condition `X3`. The state `M2` is deactivated by `X6`. In the other branch, `M3` is followed by `M4` when `X4` becomes active, and `M4` is deactivated by `X5`.

The relevant ST assignments generated from the LD representation are:

```
M4 := NOT X5 AND (M4 OR (M3 AND X4));
M3 := NOT X4 AND (M3 OR (M1 AND X3));
M2 := NOT X6 AND (M2 OR (M1 AND X2));
M1 := M1 OR (M0 AND X1);
M0 := NOT X1 AND (M0 OR X0);
```

These assignments match the state patterns defined in the conversion catalogue presented in Appendix A. The dependencies identified between the state variables are:

$$\texttt{M0} \rightarrow \texttt{M1} \rightarrow \begin{cases} \texttt{M2}, \\ \texttt{M3} \rightarrow \texttt{M4}. \end{cases}$$

The relevant distinction with respect to the previous test is that the transitions from `M1` to the two successor states are activated by different conditions. The transition to `M2` depends on `X2`, while the transition to `M3` depends on `X3`. Therefore, the two branches are represented as

alternative paths in the generated SFC, resulting in a selection divergence after the step associated with `M1`.

For this proof-of-concept test, the input conditions `X2` and `X3` are considered mutually exclusive. This assumption is important because, in the LD representation, simultaneous activation of both conditions would allow both successor states to be activated during the same scan cycle. Under the intended operating condition, only one branch is selected at a time, which matches the intended interpretation of the generated SFC selection divergence.

It should also be noted that `M1` is implemented as a self-holding state without an explicit reset condition in this test. Therefore, the purpose of this example is not to validate a complete state-reset policy, but to verify whether the implemented transformation is able to identify multiple alternative successor states originating from the same preceding state.

The SFC generated by the tool and imported into CODESYS is shown in Figure 5.8.

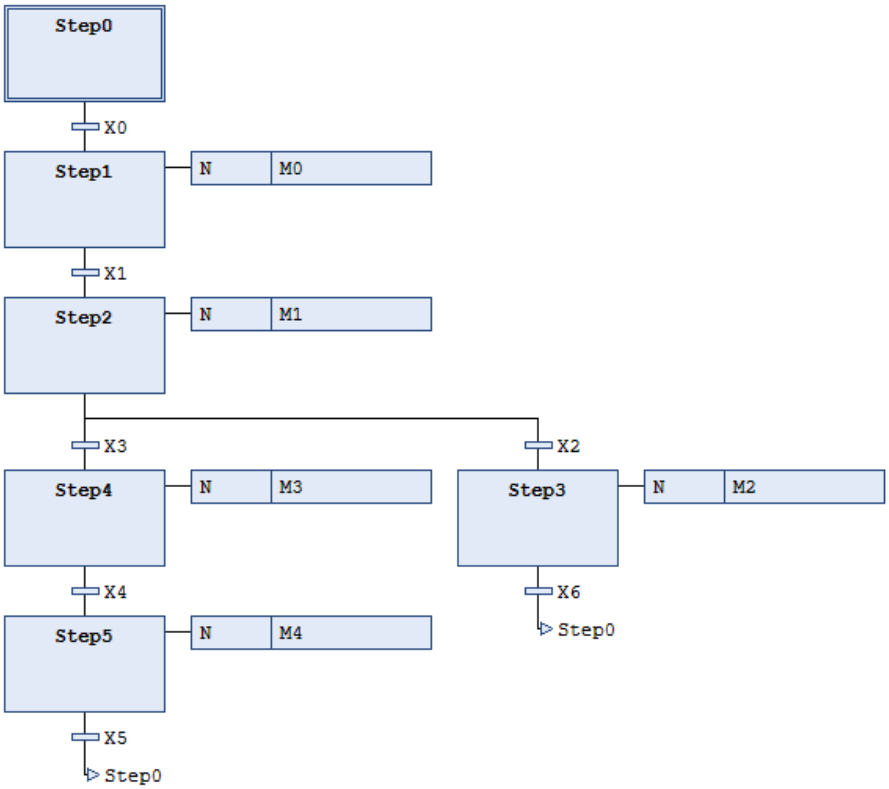


Figure 5.8: Generated SFC for Test 4 after import into CODESYS.

The generated SFC starts with the transitions `Step0` $\xrightarrow{\texttt{X0}}$ `Step1` and `Step1` $\xrightarrow{\texttt{X1}}$ `Step2`, representing the activation of `M0` and `M1`. From `Step2`, the SFC contains a selection divergence with two outgoing alternatives: condition `X2` activates the branch associated with `M2`, while condition `X3` activates the branch associated with `M3`.

On the branch associated with `M2`, condition `X6` returns the execution to `Step0`. On the other branch, condition `X4` connects the step associated with `M3` to the step associated with `M4`, and

condition `X5` subsequently returns the execution to `Step0`. This test therefore shows that the implemented transformation can structurally distinguish a selection divergence from a simultaneous divergence when multiple successor states originate from the same preceding state but are activated by distinct transition conditions. Since `M1` is intentionally represented without an explicit reset condition, the test focuses on branch classification rather than on full equivalence of the complete state-reset behavior.

5.3 Literature Case Study

Following the proof-of-concept tests, the conversion tool was applied to an example previously used in the LD-to-SFC literature. The selected case is the neutralization-tank control program introduced by Falcione and Krogh [3] and subsequently used by Lopes to evaluate his state-space-based conversion approach [9].

The example describes the operation of a tank controlled by several level and process sensors. Its actuators perform operations such as filling, mixing, heating, pH correction, and emptying the tank. The control program therefore contains several process phases, sensor-dependent transitions, and outputs that are active during particular operating states [9].

This case was selected because it provides a recognized reference from the previous literature and is more representative than the individual test programs presented in Section 5.2. It also allows the pattern-based approach developed in this dissertation to be examined using an example that was previously analyzed through a different conversion methodology. The original program is processed using the pattern catalog presented in Appendix A, and the generated SFC is then compared with the process structure identified in the previous literature.

The additional patterns required to process the retained-state equations of the example are identified in the following subsections.

5.3.1 Original LD Program

The process considered in the case study is illustrated in Figure 5.9. The tank is connected to a reservoir and a neutralizer through the inlet valves $v1$ and $v2$, respectively. The outlet valves $v3$ and $v4$ transfer the tank contents either to the next tank or back to the reservoir. The process also includes a mixer m, a heater h, level sensors $ls1$, $ls2$, and $ls3$, and the process signals as and ts.

The original LD program is shown in Figure 5.10. It contains both retained variables, whose value depends on their value in the preceding PLC scan cycle, and Boolean outputs that are directly obtained from the current input and state-variable values.

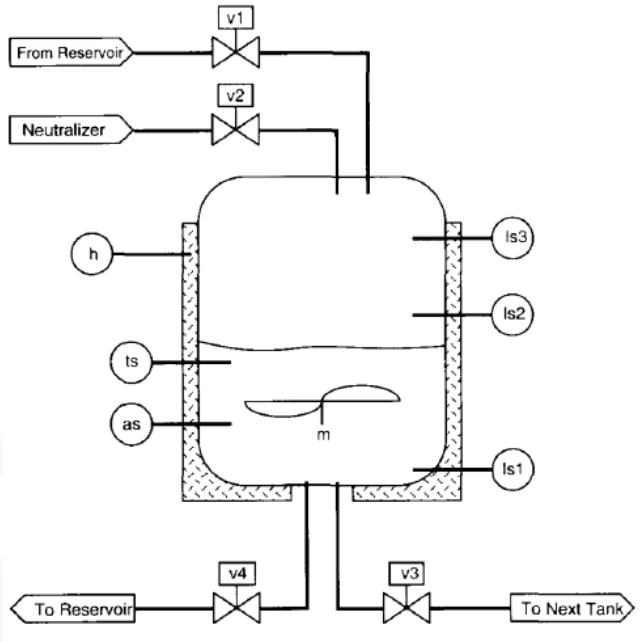


Figure 5.9: Neutralization-tank process used in the literature case study. Taken from [3]

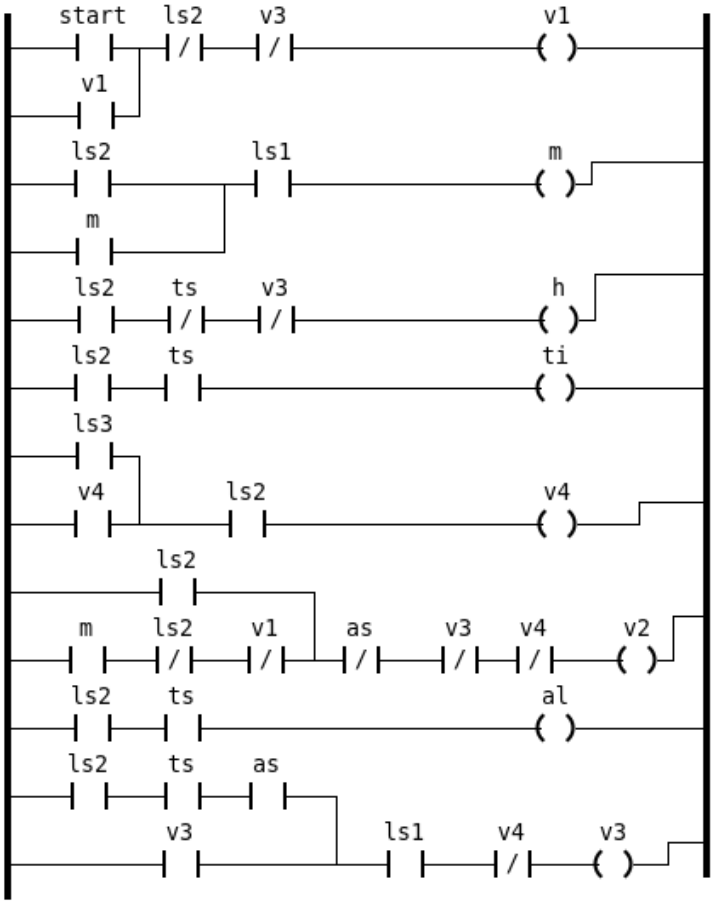


Figure 5.10: Original LD program of the neutralization-tank case study. Reproduced from [9]

The retained variables identified in the program are $v1$, m, $v4$, and $v3$. These variables contain feedback branches in their respective LD rungs and therefore preserve state between PLC scans. In contrast, h, ti, al, and $v2$ are calculated without feedback from their own previous values and are consequently treated as combinational or state-dependent outputs rather than independent process states.

The original LD program was translated into ST before being processed by the conversion tool. The relevant state-defining assignments are:

$$m := ls1 \wedge (ls2 \vee m), \tag{5.1}$$

$$v1 := \neg v3 \wedge \neg ls2 \wedge (start \vee v1), \tag{5.2}$$

$$v4 := ls2 \wedge (ls3 \vee v4), \tag{5.3}$$

$$v3 := \neg v4 \wedge ls1 \wedge \big(v3 \vee (as \wedge ts \wedge ls2)\big). \tag{5.4}$$

The remaining assignments generated from the LD program are:

$$h := \neg v3 \wedge \neg ts \wedge ls2, \tag{5.5}$$

$$ti := ts \wedge ls2, \tag{5.6}$$

$$al := ts \wedge ls2, \tag{5.7}$$

$$v2 := \neg v4 \wedge \neg v3 \wedge \neg as \wedge (ls2 \vee (\neg v1 \wedge \neg ls2 \wedge m)). \tag{5.8}$$

5.3.2 Extension of the Pattern Catalog

The patterns initially developed for the proof-of-concept tests were not sufficient to match all state-defining structures occurring in the case-study LD program. Although the program contains the same general type of memory-based LD structures considered in the previous tests, some of the state equations include additional enabling and inhibiting conditions. Since the implemented method is based on AST pattern matching, these differences must be explicitly represented in the conversion catalog.

Three additional input/output pattern pairs were therefore added to the catalog. Each pair contains an input ST pattern, which is matched against the AST generated from the LD program, and an output SFC pattern, which defines the corresponding SFC fragment. These patterns are not intended to perform Boolean simplification or equivalence checking. Instead, they describe the ST structures that are expected to result from the LD-to-ST translation of the original rungs.

Pattern $p7$: state with positive enabling condition

The first added pattern represents a self-holding state that remains active only while a positive enabling condition is true. Its ST form is:

$$s_1 := X_b \wedge (X_a \vee s_1). \tag{5.9}$$

In this pattern, s_1 is the state variable, X_a is the activation condition, and X_b is a positive enabling condition. The state becomes active when both X_a and X_b are true. Once active, the state is maintained by the feedback term s_1, but only while X_b remains true. Therefore, the corresponding entry and exit conditions are:

$$C_{\text{entry}} = X_a \wedge X_b, \tag{5.10}$$

$$C_{\text{exit}} = \neg X_b. \tag{5.11}$$

The corresponding catalog pattern is:

```
pattern_IN__ p7
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
  END_VAR

  ST_PART_BEG__
    s1 := Xb AND (Xa OR s1);
  ST_PART_END__
end_pattern__
```

The generated SFC fragment contains a transition from the source step to the state step when $X_a \wedge X_b$ is true, an action associated with s_1, and an exit transition when $\neg X_b$ becomes true:

```
pattern_OUT__ p7
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := Xa AND Xb;
  END_TRANSITION

  STEP StepTo:
    s1(N);
```

```
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := NOT Xb;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

This pattern is used in the case study for the variables m and $v4$. For m, the mapping is $s_1 = m$, $X_a = ls2$, and $X_b = ls1$. For $v4$, the mapping is $s_1 = v4$, $X_a = ls3$, and $X_b = ls2$.

Pattern $p8$: state with two inhibiting conditions

The second added pattern represents a self-holding state with two inhibiting conditions. Its ST form is:

$$s_1 := \neg X_c \wedge \neg X_b \wedge (X_a \vee s_1). \tag{5.12}$$

In this case, X_a activates the state, while X_c and X_b prevent the state from being active. The state is therefore entered only when the activation condition is true and both inhibiting conditions are false:

$$C_{\text{entry}} = X_a \wedge \neg X_b \wedge \neg X_c, \tag{5.13}$$

$$C_{\text{exit}} = X_c \vee X_b. \tag{5.14}$$

The corresponding input pattern is:

```
pattern_IN__ p8
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
  END_VAR

  ST_PART_BEG__
    s1 := NOT(Xc) AND NOT(Xb) AND (Xa OR s1);
  ST_PART_END__
end_pattern__
```

The output pattern generates an SFC state associated with s_1. The transition into the state preserves the activation and inhibition logic, while the exit transition is enabled as soon as either inhibiting condition becomes true:

```
pattern_OUT__ p8
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := Xa AND NOT Xb AND NOT Xc;
  END_TRANSITION

  STEP StepTo:
    s1(N);
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := Xc OR Xb;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

This pattern is used to represent the assignment to $v1$. In the case-study program, the corresponding mapping is $s_1 = v1$, $X_a = start$, $X_b = ls2$, and $X_c = v3$. The entry condition is therefore $start \wedge \neg ls2 \wedge \neg v3$, and the exit condition is $v3 \vee ls2$.

Pattern $p9$: guarded self-holding state with compound activation condition

The third added pattern represents a self-holding state whose activation and memory term are both guarded by the same enabling expression. Its ST form is:

$$s_1 := \neg X_e \wedge X_d \wedge \big(s_1 \vee (X_a \wedge X_b \wedge X_c)\big). \tag{5.15}$$

In this expression, s_1 is the state variable, $X_a \wedge X_b \wedge X_c$ is the activation condition, and $\neg X_e \wedge X_d$ is the holding guard. The state can therefore only be activated when the full expression

$$C_{\text{entry}} = \neg X_e \wedge X_d \wedge X_a \wedge X_b \wedge X_c \tag{5.16}$$

is true. Once active, the feedback term s_1 maintains the state while X_d remains true and X_e remains false. The state is deactivated when this guard is no longer satisfied, giving the exit condition:

$$C_{\text{exit}} = \neg X_d \vee X_e. \tag{5.17}$$

The corresponding input pattern is:

```
pattern_IN__ p9
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
    Xd : BOOL;
    Xe : BOOL;
  END_VAR

  ST_PART_BEG__
    s1 := NOT(Xe) AND Xd AND (s1 OR (Xa AND Xb AND Xc));
  ST_PART_END__
end_pattern__
```

The output pattern generates a transition into the state when the complete guarded activation condition is true. The state is then represented as an SFC step with an action associated with s_1, and the exit transition is generated from the negation of the holding guard:

```
pattern_OUT__ p9
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
    Xd : BOOL;
    Xe : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := NOT Xe AND Xd AND Xa AND Xb AND Xc;
  END_TRANSITION

  STEP StepTo:
    s1(N);
  END_STEP
```

```
  TRANSITION FROM StepTo TO StepExit
    := NOT Xd OR Xe;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

This pattern is used to represent the state variable $v3$. In the case study, the mapping is $s_1 = v3$, $X_a = as$, $X_b = ts$, $X_c = ls2$, $X_d = ls1$, and $X_e = v4$. This gives:

$$v3 := \neg v4 \wedge ls1 \wedge \big(v3 \vee (as \wedge ts \wedge ls2)\big), \tag{5.18}$$

with the corresponding entry and exit conditions:

$$C_{\text{entry}} = \neg v4 \wedge ls1 \wedge as \wedge ts \wedge ls2, \tag{5.19}$$

$$C_{\text{exit}} = \neg ls1 \vee v4. \tag{5.20}$$

Table 5.1 summarizes the mappings identified in the case-study program.

Table 5.1: Mapping between case-study state variables and the added patterns.

Variable	Pattern	Entry condition	Exit condition
$v1$	$p8$	$start \wedge \neg ls2 \wedge \neg v3$	$v3 \vee ls2$
m	$p7$	$ls2 \wedge ls1$	$\neg ls1$
$v4$	$p7$	$ls3 \wedge ls2$	$\neg ls2$
$v3$	$p9$	$\neg v4 \wedge ls1 \wedge as \wedge ts \wedge ls2$	$\neg ls1 \vee v4$

The variables m, $v1$, $v3$, and $v4$ are therefore treated as state-defining variables because their assignments contain feedback terms and can be mapped to SFC steps. By contrast, the remaining assignments do not contain self-holding structures and do not define independent states:

$$h := \neg v3 \wedge \neg ts \wedge ls2, \tag{5.21}$$

$$ti := ts \wedge ls2, \tag{5.22}$$

$$al := ts \wedge ls2, \tag{5.23}$$

$$v2 := \neg v4 \wedge \neg v3 \wedge \neg as \wedge \big(ls2 \vee (\neg v1 \wedge \neg ls2 \wedge m)\big). \tag{5.24}$$

These expressions describe outputs or intermediate variables whose values depend on process inputs and, in some cases, on more than one state variable. The existing `StepOn` mechanism is not appropriate for these assignments because it assumes that the right-hand side contains a single identifiable state variable to which the resulting action can be attached.

For this reason, the assignments to h, ti, al, and $v2$ were excluded from the final state-pattern conversion. Attaching them to only one SFC step would omit part of the original enabling condition and could change the program behavior. Their correct treatment requires a more general action-generation mechanism capable of preserving the complete Boolean expression and associating the resulting assignment with the appropriate active state configuration. This remains outside the scope of the current implementation and is identified as a limitation of the current pattern catalog.

5.3.3 Generated SFC

The four state-defining assignments were matched and converted into the SFC shown in Figure 5.11.

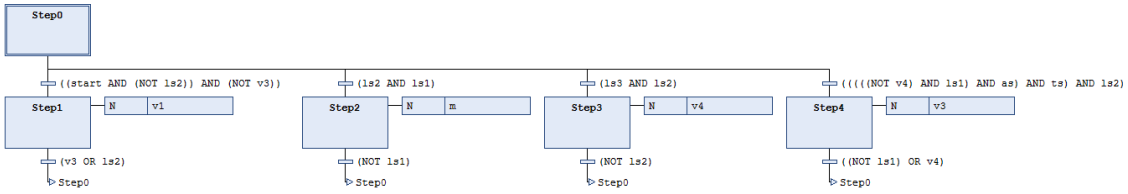


Figure 5.11: SFC generated for the retained-state patterns detected in the case-study program.

The initial step, `Step0`, is introduced by the top-level output pattern and acts as the common root of the generated SFC. It is therefore a structural element created by the conversion process and should not necessarily be interpreted as a physical state of the original process.

Each matched assignment is initially converted independently into a local SFC fragment composed of an entry transition, a step with an associated action, and an exit transition. During the composition of these fragments, an explicit dependency on another recognized state-defining variable may be used to identify the predecessor step. When no such predecessor relation is established, the fragment is connected to the default root, `Step0`.

In this case, the activation conditions of the four detected variables do not provide an explicit predecessor relation that the current composition method can directly convert into a connection between generated steps. References to negated state variables, such as $\neg v3$ in the activation condition of $v1$ or $\neg v4$ in the activation condition of $v3$, remain part of the corresponding Boolean transition guards. They are not yet interpreted as structural relations between the respective SFC steps. Consequently, all four local fragments are initially connected to `Step0`. Their order in the generated representation follows the order in which the assignments are processed, but this order does not by itself establish a state-predecessor relation.

Since four transitions leave the same source step, the XML generation stage must introduce a divergence. The current implementation classifies this divergence by comparing the transition conditions. When the outgoing transitions share the same activation condition, they are interpreted as simultaneous branches. When their conditions differ, they are represented using a selection

divergence. In the case-study output, the four activation conditions are different, and the branches are therefore placed after a selection divergence.

This classification is based on the local transition expressions available during generation. It does not constitute a global verification that the four states are mutually exclusive. As discussed in Section 5.4, some of the conditions may become true concurrently, while other dependencies should be promoted from Boolean terms into explicit selection or simultaneous structures.

Each outgoing path corresponds to one state variable detected in the LD program:

- `Step1` activates $v1$;
- `Step2` activates m;
- `Step3` activates $v4$;
- `Step4` activates $v3$.

The transition leading to each step is obtained from the activation component of the corresponding feedback assignment. Likewise, the transition placed after each step is derived from the condition that disables the associated state variable.

For example, Equation (5.2) can be separated into an activation condition and a holding condition. The variable $v1$ is activated when

$$start \wedge \neg ls2 \wedge \neg v3 \tag{5.25}$$

is true. Once active, it remains active while both $ls2$ and $v3$ remain false. Its SFC exit condition is therefore

$$ls2 \vee v3. \tag{5.26}$$

The same interpretation is applied to the remaining state variables. When the current composition stage does not identify a subsequent state associated with an exit condition, the branch returns to the common root through a jump to `Step0`. The jump is preceded by the corresponding exit transition, ensuring that the step remains active until its deactivation condition becomes true.

The resulting local structures are therefore:

$$\begin{aligned}
&\texttt{Step0} \xrightarrow{start\wedge\neg ls2\wedge\neg v3} \texttt{Step1}/v1 \xrightarrow{v3\vee ls2} \texttt{Step0},\\
&\texttt{Step0} \xrightarrow{ls2\wedge ls1} \texttt{Step2}/m \xrightarrow{\neg ls1} \texttt{Step0},\\
&\texttt{Step0} \xrightarrow{ls3\wedge ls2} \texttt{Step3}/v4 \xrightarrow{\neg ls2} \texttt{Step0},\\
&\texttt{Step0} \xrightarrow{\neg v4\wedge ls1\wedge as\wedge ts\wedge ls2} \texttt{Step4}/v3 \xrightarrow{\neg ls1\vee v4} \texttt{Step0}.
\end{aligned}$$

5.4 Discussion of Results

The simpler test cases confirmed the basic operation of the proposed conversion method. The results showed that the tool can identify assignments corresponding to state-defining variables, generate associated SFC steps, derive entry and exit transitions, attach actions, and represent structural elements such as selection divergences, simultaneous divergences, and jumps in PLCopen XML.

The literature case study provided a more demanding and complex validation scenario. Unlike the proof-of-concept tests, the case study contains several retained variables, non-state-defining assignments, and dependencies between process states and sensor conditions. Its processing therefore required the extension of the pattern catalog to recognize the feedback equations associated with m, $v1$, $v3$, and $v4$.

These variables were treated as state-defining because their assignments contain feedback references and therefore depend on values preserved from a previous PLC scan.

The assignments to h, ti, al, and $v2$ were not converted into additional SFC steps because they do not independently retain state. Nevertheless, they remain part of the behavior of the original LD program. A fully equivalent SFC would therefore have to preserve them as step actions, conditional actions, or auxiliary ST code associated with the appropriate SFC locations. Since the current implementation does not yet provide a generalized method for placing non-state-defining assignments, the generated result should be interpreted as a reduced SFC representation containing the principal memory-dependent behavior rather than a complete replacement of the original LD program.

An obvious limitation is that the conversion methodology depends directly on the contents of the `pattern_file` catalog. Each assignment to be converted must match a defined input pattern and must have a corresponding output transformation. Assignments for which either the detection pattern or the associated SFC fragment is missing cannot be included in the generated result. Consequently, the coverage of the converter is determined not only by the pattern-matching algorithm, but also by the completeness and scalability of the pattern catalog.

The case study also demonstrated that the recognition of each state-defining assignment independently is not sufficient to reconstruct the complete semantically equivalent SFC. The individual patterns expose local state behavior, but the resulting fragments must subsequently be composed according to the dependencies and possible simultaneous activations of all state variables.

For example, the mixer variable m becomes active when the liquid level reaches $ls2$ and remains active until the level falls below $ls1$. During this interval, the principal process sequence may progress through other states. Therefore, m is better represented as a branch of simultaneous divergence rather than as a step placed sequentially on the main path of the process. A simultaneous divergence may activate both the mixer branch and the principal process branch when $ls2$ becomes true.

The principal branch itself requires a selection between the behavior associated with $v3$ and that associated with $v4$. In normal process operation, $v3$ represents product discharge after the

quality conditions *as* and *ts* are met, while $v4$ is activated when the level reaches $ls3$. These behaviors should therefore originate from a common treatment state through alternative transitions. More generally, when the entry condition of one state-defining variable contains the negation of another state-defining variable, the conversion algorithm should identify the resulting mutual-exclusion dependency and represent the corresponding steps through a selection divergence, provided that they originate from the same preceding process state.

Another relevant result concerns the preservation of the PLC scan execution order. The ST program generated from the LD executes assignments sequentially. Consequently, an assignment located later in the program may read the value of a variable already updated earlier during the same scan. This occurs for $v3$, whose behavior is more accurately represented as

$$v3^{k+1} = \neg v4^{k+1} \wedge ls1^{k} \wedge \left[\left(as^{k} \wedge ts^{k} \wedge ls2^{k} \right) \vee v3^{k} \right]. \tag{5.27}$$

The use of $v4^{k+1}$, rather than $v4^{k}$, reflects the fact that the assignment to $v4$ is executed before the assignment to $v3$ during the PLC scan. Therefore, when the right-hand side of the assignment to $v3$ is evaluated, it already depends on the value of $v4$ computed in the current scan. The feedback term of $v3$, on the other hand, still refers to the previous value of $v3$, since the new value of $v3$ has not yet been assigned.

The local behavior extracted for $v3$ therefore preserves the scan-order dependency on $v4$. However, in the current implementation, this dependency is preserved only as part of the Boolean transition condition. Although the generated SFC places the steps associated with $v3$ and $v4$ in different branches of a selection divergence, this structure results from the comparison of outgoing transition conditions from `Step0`, not from a global semantic analysis of the interaction between the two retained variables. Consequently, the limitation lies in the composition of the independently generated SFC fragments: the current implementation does not yet determine whether such branches represent true alternatives, concurrent behaviors, or process-dependent relations that should be reorganized using additional dependency information.

Dependencies between different assignments are currently interpreted mainly from the order in which the generated ST statements are analyzed and from explicit references to recognized state-defining variables. When the entry condition of a detected state contains another state-defining variable, that variable can be used to establish a predecessor relation. Otherwise, the relative position of the corresponding assignments remains the principal ordering information available to the current composition stage.

For programs with more complex interactions, such as the case study considered here, this local and order-based composition is insufficient to determine all concurrency, mutual-exclusion, and synchronization relations. A possible extension would therefore be to complement the AST pattern-matching stage with a global analysis methodology already explored in previous LD-to-SFC approaches, such as state modelling or graph-based dependency analysis. Each matched AST subtree could first be converted into a local SFC fragment, after which the dependencies between these fragments would be represented in a higher-level AST structure. This dependencies repre-

sentation could then be used to determine predecessor relations, alternative branches, simultaneous activations, and convergence points before generating the final AST SFC.

The manual analysis also identified opportunities to simplify some transition conditions using information about the physical process. In the tank, the level sensors are arranged vertically, so reaching ls3 necessarily means that ls2 and ls1 are also active, while reaching ls2 also implies that ls1 is active. As a result, conditions such as ls2 together with ls1, or ls3 together with ls2, contain redundant terms.

The same principle applies to the transition associated with v3. If the source SFC step already represents a situation in which the level has reached ls2 and v4 is inactive, those conditions do not need to be repeated in the transition guard. In that context, the transition only needs to verify that the quality conditions as and ts are satisfied and that the level has not yet reached ls3.

These simplifications depend on the physical arrangement of the sensors and on the conditions already implied by the active SFC step. They should therefore be treated as plant-dependent semantic post-processing rather than as transformations that can be derived from the LD syntax alone.

The generated SFC represents the locally identified state transitions, but it does not yet reconstruct the complete global state structure of the LD program. Some activation conditions may become true during the same PLC scan, causing multiple state-defining variables to become active concurrently. Branches initially represented as alternatives may require nested simultaneous divergences, while other states may be mutually exclusive and require selection divergences.

Achieving full semantic equivalence does not require abandoning the pattern-based approach, but it does require an additional global composition stage. Such a stage could use dependency analysis, explicit state-space exploration, or a combination of these techniques to determine which detected states are mutually exclusive, which may coexist, where synchronization is necessary, and how non-state-defining assignments should be associated with the resulting SFC steps.

Overall, the results confirm that AST pattern matching is effective in identifying which LD constructs and variables correspond to specific SFC structures. The design patterns establish a direct mapping between an LD construct, its equivalent ST representation, and the corresponding SFC fragment, allowing the semantics of each individual assignment to be preserved throughout the conversion. The main limitation does not therefore lie in the translation of the assignments themselves, but in determining how the generated SFC structures are related at the global program level when their behavior depends on the values of different variables. When the relation is expressed through the same state variable, or through an explicit dependency on another state-defining variable, the current implementation can already generate the corresponding divergences and convergences. More complex relations involving several variables, implicit concurrency, or mutual exclusion still require a higher-level analysis of the complete program.

Chapter 6

Conclusion

This dissertation addressed the automatic conversion of Ladder Diagram (LD) programs into Sequential Function Chart (SFC) representations within the scope of IEC 61131-3. The work was motivated by the widespread use of LD in industrial automation and by the difficulty of identifying sequential behavior when it is distributed across feedback equations, Boolean conditions, and several program rungs.

The literature review showed that previous LD-to-SFC conversion methods generally follow graph-based, logic-based, or state-space-based approaches. These methods can recover relevant control information, but they may depend on restrictive assumptions, program-specific rules, or extensive exploration of possible program states. This motivated the investigation of a structural alternative based on recurring LD programming patterns.

The developed methodology uses Structured Text as an intermediate representation and applies pattern matching over its Abstract Syntax Tree. LD structures (rungs) that define states are identified through their feedback expressions and are mapped to SFC steps, transitions, actions, potential divergences/ convergences, and jumps. This approach separates the recognition of local program structures from the generation of the corresponding SFC fragments and allows the conversion catalog to be extended through additional input and output patterns.

A proof-of-concept converter was implemented and evaluated through feature-oriented examples and a case study taken from the LD-to-SFC literature. The tests confirmed that the approach can detect several forms of state-defining assignments and produce importable PLCopen XML representations. The case study also showed that new LD structures can be supported by extending the pattern catalog without redesigning the complete conversion process.

Nevertheless, the current methodology has two important limitations. First, the conversion depends directly on the available pattern catalog. Each assignment must match a defined detection pattern and must have a corresponding SFC output structure. Programs containing unsupported expressions cannot therefore be converted completely.

Second, the independent analysis of each AST assignment is insufficient to reconstruct every relation between the rungs of the complete program. It can determine the entry and exit conditions of an individual state, but it cannot always establish whether different states are mutually exclusive

or may become active simultaneously, especially if this semantics are represented through dependencies of non state-defining rungs. This information is necessary to decide the correct sequence of selection and simultaneous divergences in the final SFC.

The relevance of the proposed conversion process lies in the structural information it introduces before any global behavioral analysis is performed. Each detected LD construct is first translated into its equivalent AST subtree and then mapped to a specific SFC fragment. The resulting fragments are therefore not abstract states obtained only from combinations of Boolean variables; they already carry explicit SFC meaning, including candidate steps, actions, entry and exit transitions, and possible predecessor relations. Furthermore, because the matched subtrees can themselves be organized within a higher-level, dependency-relation representation, the conversion process provides an initial structural model of the program rather than a collection of unrelated local matches.

This structural model can help bridge the gap between the pattern-oriented method and previously proposed graph-based or state-space-based methodologies. Instead of applying those methods directly to the complete LD program, they may operate over the smaller set of state-related subtrees and SFC fragments already extracted by pattern matching. Graph analysis may then be used to identify dependencies between the fragments, while state modelling may determine which of them are mutually exclusive, concurrent, reachable, or synchronized. In this way, the AST-based conversion contributes the structural basis of the final SFC, while complementary methodologies provide the global behavioral relations that cannot always be inferred from the local patterns alone.

Such a hybrid approach preserves the main advantage of the pattern-oriented method: the direct reuse of known LD structures. At the same time, it addresses the main semantic limitation of local pattern analysis by introducing global behavioral verification only where it is required.

Overall, this dissertation demonstrates that AST-based pattern recognition provides a practical foundation for LD-to-SFC conversion. Although the current implementation does not yet provide complete semantic reconstruction for every LD program, it establishes a modular conversion framework in which structural detection and behavioral analysis can be combined. This creates a clear path towards more complete, maintainable, and reusable conversion tools for IEC 61131-3 control software.

6.1 Future Work

Further work should extend the pattern language with dedicated constructs for additional LD and SFC elements, including function and function-block calls, timers, counters, edge-detection elements, set/reset behavior, action qualifiers, and multiple actions associated with the same step. These constructs would allow the catalog to represent more complex control structures directly, without relying on large generic subtree patterns.

The pattern catalog should also support verification of matched AST subtrees and hierarchical pattern composition. Allowing a `PATTERN_N__` node to contain other `PATTERN_N__` nodes would promote reuse, reduce catalog size, and simplify the definition of more complex structures.

Before assembling the final SFC AST, the matched subtrees should be processed by a complementary global-analysis method, such as dependency-graph analysis or reduced state modelling. This stage would determine precedence, concurrency, mutual exclusion, divergence, and convergence relations between the generated SFC fragments.

A final verification stage should be introduced before output generation to validate the resulting PLCopen XML against the corresponding schema and to check structural consistency, including valid identifiers, references, connections, step-transition ordering, divergences, convergences, jumps, and action associations.

The methodology should also be evaluated in combination with complementary approaches, such as dependency-graph analysis and reduced state-space modelling, to assess how pattern-derived AST subtrees can support the construction of the final SFC structure.

6.2 Research Article

As part of this dissertation, a research article entitled *Translating IEC 61131-3 LD Programs to SFC – A Pattern Matching Approach* was prepared to disseminate the preliminary results of the proposed LD-to-SFC conversion approach [14].

The article presents the motivation for converting LD programs into SFC representations and describes the principal elements of the proposed method, including the use of ST as an intermediate representation, AST-based pattern matching, and the generation of SFC programs in PLCopen XML format. Since the article was prepared before the completion of the dissertation, it documents an earlier version of the conversion tool and therefore does not include the subsequent extensions to the conversion pipeline or the complete set of tests presented in this work.

References

[1] *Beremiz: Open Source Automation Software*. http://www.beremiz.org/. Accessed: 2026-01-04.

[2] CODESYS Group. *CODESYS Development System*. https://www.codesys.com/products/engineering/development-system/. Accessed: 2026-06-20.

[3] Anthony Falcione and Bruce H. Krogh. "Design Recovery for Relay Ladder Logic". In: *IEEE Control Systems Magazine* 13.2 (1993), pp. 90–98. DOI: 10.1109/37.206160.

[4] *IEC 61131-10: PLCopen XML Exchange Format*. https://www.plcopen.org/standards/logic/iec-61131-10/. Accessed: 2026-06-18.

[5] International Electrotechnical Commission. *IEC 61131-3: Programmable Controllers — Part 3: Programming Languages*. 3rd. Geneva, Switzerland: IEC, 2013.

[6] International Electrotechnical Commission. *IEC 61131-3:2025: Programmable Controllers – Part 3: Programming Languages*. https://webstore.iec.ch/en/publication/68533. Edition 4.0, published 22 May 2025. Accessed: 2026-06-26. Geneva, Switzerland, 2025.

[7] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. 2nd ed. Berlin, Heidelberg: Springer, 2010. DOI: 10.1007/978-3-642-12015-2.

[8] Robert W. Lewis. *Programming Industrial Control Systems Using IEC 1131-3*. IEE Control Engineering Series 50. IET, 1998.

[9] V. E. Lopes. "Converting LD to SFC (IEC 61131-3)". MSc Dissertation. Porto, Portugal: Faculty of Engineering, University of Porto (FEUP), 2017.

[10] R. K. Vimal Nandhan and N. Ramesh Babu. "Understanding of Logic in Ladder Program with Its Transformation into Sequential Graph Using State Space-Based Approach". In: *International Journal of Mechatronics and Manufacturing Systems* 6.2 (2013), pp. 159–182. DOI: 10.1504/IJMMS.2013.053827.

[11] *PLCopen*. https://www.plcopen.org/. Accessed: 2026-06-18.

[12] Kamil Rosiak et al. "Custom-Tailored Clone Detection for IEC 61131-3 Programming Languages". In: *Journal of Systems and Software* 175 (2021), pp. 110–122. DOI: 10.1016/j.jss.2021.111070. URL: https://doi.org/10.1016/j.jss.2021.111070.

[13] Mario de Sousa et al. *MATIEC: Compiler for IEC 61131-3 Programming Languages*. URL: https://github.com/beremiz/matiec (visited on 06/20/2026).

[14] Mário de Sousa, Duarte Tavares, and Yimin Zhang. "Translating IEC 61131-3 LD Programs to SFC: A Pattern Matching Approach". Manuscript submitted for publication. 2026.

[15] Toshio Zanma, Shinji Doki, and Shigeru Okuma. “Extraction of Action Sequences from Ladder Programs Using Temporal Logic”. In: *IEEE Transactions on Industrial Electronics* 53.4 (2006), pp. 1289–1298. DOI: 10.1109/TIE.2006.878313.

Appendix A

Conversion Pattern Catalogue

This appendix presents the complete conversion-pattern catalog used by the implemented tool and throughout the evaluation described in Chapter 5.

To add a conversion pattern, the user defines a pair of declarations named `pattern_IN__` and `pattern_OUT__`, both sharing a unique pattern name and ending with `end_pattern__`. The simplest approach is to adapt an existing pair from the catalog:

1. Declare the required variables in each `VAR` block. Their names act as placeholders that are replaced by the identifiers captured from the source program.
2. Define the ST structure to match between `ST_PART_BEG__` and `ST_PART_END__`. Matching is structural, so operator order and expression grouping must follow the ST generated from the LD program.
3. Define the corresponding SFC fragment between `SFC_PART_BEG__` and `SFC_PART_END__`. Use `StepFrom` and `StepTo` for state fragments, or `StepOn` to attach an action to an existing state step.
4. Add the new pattern name to the `PATTERN_N__` lists in both declarations of the top-level pattern `AAA`.

The updated file is selected through the Stage 4 option `-O p=<filename>`. Each new pattern should be tested with a representative program to verify that the generated SFC preserves the intended behavior.

Listing A.1 presents the complete catalog.

Listing A.1: Conversion-pattern catalogue used during the evaluation.

```
pattern_IN__ AAA
  VAR
    x : BOOL;
  END_VAR
```

```
  PATTERN_N__(p1, p2, p3, p4, p5, p6, p7, p8 ,p9)
end_pattern__



pattern_OUT__ AAA
  VAR
    foo : BOOL;
  END_VAR

  INITIAL_STEP Step0:
  END_STEP

  PATTERN_N__(p1, p2, p3, p4, p5, p6, p7, p8 ,p9)
end_pattern__


(*************************************************)
(*************************************************)
(*************************************************)


pattern_IN__ p1
  VAR
    Mx  : BOOL;
    Mps : BOOL;
    Xa  : BOOL;
    Xb  : BOOL;
  END_VAR

  ST_PART_BEG__
    Mx := (Mx AND NOT(Xb)) OR (Mps AND Xa);
  ST_PART_END__
end_pattern__



pattern_OUT__ p1
  VAR
```

```
    Mx   : BOOL;
    Mps  : BOOL;
    Xa   : BOOL;
    Xb   : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := Xa and Mps;
  END_TRANSITION

  STEP StepTo:
    Mx(N);
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := Xb;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

```
pattern_IN__ p2
  VAR
    Mx   : BOOL;
    Mps  : BOOL;
    Xa   : BOOL;
    Xb   : BOOL;
  END_VAR

  ST_PART_BEG__
    Mx := NOT(Xb) AND (Mx OR (Mps AND Xa));
  ST_PART_END__
end_pattern__
```

```
pattern_OUT__ p2
  VAR
    Mx  : BOOL;
    Mps : BOOL;
    Xa  : BOOL;
    Xb  : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := Xa;
  END_TRANSITION

  STEP StepTo:
    Mx(N);
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := Xb;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

```
pattern_IN__ p3
  VAR
    zz, kk : BOOL;
    s1 : BOOL;
  END_VAR
  ST_PART_BEG__
    s1 := NOT(kk) AND (s1 OR zz);
  ST_PART_END__
```

```
end_pattern__
```

```
pattern_OUT__ p3
  VAR
    zz, kk : BOOL;
    s1 : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := zz;
  END_TRANSITION

  STEP StepTo:
    s1(N);
  END_STEP


  TRANSITION FROM StepTo TO StepExit
    := kk;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

```
pattern_IN__ p4
  VAR
    zz, kk : BOOL;
    s1 : BOOL;
  END_VAR
  ST_PART_BEG__
    s1 := kk AND (s1 OR NOT(zz));
```

```
  ST_PART_END__
end_pattern__
```

```
pattern_OUT__ p4
  VAR
    zz, kk : BOOL;
    s1 : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := kk;
  END_TRANSITION

  STEP StepTo:
    s1(N);
  END_STEP
  SFC_PART_END__
end_pattern__
```

```
pattern_IN__ p5
  VAR
    kk : BOOL;
    ba, bb : BOOL;
  END_VAR
  ST_PART_BEG__
    bb := bb OR (ba AND kk);
  ST_PART_END__
end_pattern__
```

```
pattern_OUT__ p5
  VAR
    kk : BOOL;
    ba, bb : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := kk;
  END_TRANSITION

  STEP StepTo:
    bb(N);
  END_STEP
  SFC_PART_END__
end_pattern__
```

```
pattern_IN__ p6
  VAR
    dd : BOOL;
    s2 : BOOL;
  END_VAR
  ST_PART_BEG__
    s2 := dd;
  ST_PART_END__
end_pattern__
```

```
pattern_OUT__ p6
  VAR
    dd : BOOL;
    s2 : BOOL;
```

```
  END_VAR

  SFC_PART_BEG__
  STEP StepOn:
    s2(N);
  END_STEP
  SFC_PART_END__
end_pattern__




(************************************)
(******** CASE STUDY PATTERNS ********)
(************************************)



pattern_IN__ p7
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
  END_VAR

  ST_PART_BEG__
    s1 := Xb AND (Xa OR s1);
  ST_PART_END__
end_pattern__


pattern_OUT__ p7
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
```

```
    := Xa AND Xb;
  END_TRANSITION

  STEP StepTo:
    s1(N);
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := NOT Xb;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

```
pattern_IN__ p8
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
  END_VAR

  ST_PART_BEG__
    s1 := NOT(Xc) AND NOT(Xb) AND (Xa OR s1);
  ST_PART_END__
end_pattern__
```

```
pattern_OUT__ p8
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
```

```
    := Xa AND NOT Xb AND NOT Xc;
  END_TRANSITION

  STEP StepTo:
    s1(N);
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := Xb OR Xc;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```

```
pattern_IN__ p9
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
    Xd : BOOL;
    Xe : BOOL;
  END_VAR

  ST_PART_BEG__
    s1 := NOT(Xe) AND Xd AND (s1 OR (Xa AND Xb AND Xc));
  ST_PART_END__
end_pattern__
```

```
pattern_OUT__ p9
  VAR
    s1 : BOOL;
    Xa : BOOL;
    Xb : BOOL;
    Xc : BOOL;
    Xd : BOOL;
```

```
    Xe : BOOL;
  END_VAR

  SFC_PART_BEG__
  TRANSITION FROM StepFrom TO StepTo
    := NOT Xe AND Xd AND Xa AND Xb AND Xc;
  END_TRANSITION

  STEP StepTo:
    s1(N);
  END_STEP

  TRANSITION FROM StepTo TO StepExit
    := Xe OR NOT Xd;
  END_TRANSITION
  SFC_PART_END__
end_pattern__
```