FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Transformation of GRAFCET Specifications into IEC 61131-3 Code

André Filipe Fernandes de Azevedo Osório



Master's in Electrical and Computer Engineering

Supervisor: Prof. Paulo José Lopes Machado Portugal

July 31, 2026

© André Filipe Fernandes de Azevedo Osório, 2026

Abstract

In industrial automation, GRAFCET (IEC 60848), among many other tools and standards, is employed to formally specify sequential control logic. Since Programmable Logic Controllers (PLCs) execute control programs written in IEC 61131-3 languages rather than GRAFCET directly, engineers must manually translate graphical specifications into executable code. Owing to semantic differences between the two representations and the absence of a standardized automated transformation process, this translation can introduce inconsistencies between the original control specification and the implemented PLC logic.

This dissertation proposes and implements a toolchain that automates the transformation of GRAFCET specifications created in GRAFCET-Studio into IEC 61131-3 code, compliant with the PLCOpenXML file structure. To achieve this, the work formalizes a GRAFCET meta model, defines translation rules for mapping GRAFCET constructs to IEC 61131-3 and builds support for OPC-UA exposure of relevant variables.

The proposed approach begins by extracting the information from a GRAFCET-Studio save file, which had to first be interpreted, as components do not discriminate their interconnections, and then generating the IEC 61131-3 code, more specifically Structured Text, implanting it onto the PLCOpenXML project file template.

With this in mind, multiple test scenarios were created, imported to the CODESYS development environment and extensively tested to ensure the semantic correctness of the generated control logic and the OPC related variables, thus validating the toolchain's implementation.

The findings acquired from these tests indicate that the toolchain developed can fully translate any GRAFCET component and structure to IEC 61131-3 (Structured Text), within the scope of the evaluated tests. Moreover, the variables, exposed via the integrated CODESYS OPC-UA server, were all functional and were correctly represented. Furthermore, internal tool validation confirms that the meta model succeeds in formatting the components to a data model more closely tied to the GRAFCET standard. Additionally, the algorithm developed to extract the information from the GRAFCET-Studio save file, also successfully creates said components and makes the appropriate connection arcs between each other as expected.

Therefore, the proposed approach is a viable method for automating the translation of GRAFCET specifications into IEC 61131-3 programs. By preserving the semantics of the original GRAFCET specification throughout the transformation process, it enables reliable automatic generation of executable PLC programs while reducing development effort and the likelihood of implementation errors. Furthermore, the proposed toolchain demonstrates the applicability of Model-Driven Software Engineering principles to industrial automation and establishes a flexible foundation for future extensions, including support for additional engineering tools and formal verification.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

The work of this dissertation contributes primarily to SDG 9 (Industry, Innovation and Infrastructure) and SDG 8 (Decent Work and Economic Growth) by advancing model-driven engineering methodologies for industrial automation. The proposed automatic transformation of GRAFCET specifications into IEC 61131-3 code enhances industrial software reliability, reduces engineering time, minimizes human error and supports digitization within Industry 4.0 environments.

Additionally, it also supports SDG 4 (Quality Education) through knowledge formalization and dissemination in automation engineering, and SDG 12 (Responsible Consumption and Production) by improving software maintainability and life cycle efficiency of industrial control systems.

The specific SDGs and targets addressed by this work are listed below:

SDG 4 Ensure inclusive and equitable quality education and promote lifelong learning opportunities for all

SDG 8 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 12 Ensure sustainable consumption and production patterns

SGD	Target	Contribution	Performance Indicators and Metrics
9	9.4	The automatic transformation of GRAFCET specifications into IEC 61131-3 code improves the efficiency, reliability, and maintainability of industrial automation systems, supporting the modernization of industrial control infrastructures.	Reduction in manual PLC programming effort; decrease in implementation errors; improvement in software maintainability metrics.
	9.5	The proposed toolchain fosters technological innovation in industrial software engineering by applying model-driven engineering and formalized transformations between automation standards.	Validation through industrial case studies; correctness and semantic equivalence of generated PLC code.
	9.b	The use of standardized automation languages and interoperable toolchains supports the development and deployment of advanced industrial technologies aligned with Industry 4.0 principles.	Successful integration with GRAFCET-Studio and CODESYS; compliance with IEC 61131-3 standards.
8	8.2	Automation of PLC code generation increases engineering productivity by reducing development time and minimizing repetitive and error-prone manual tasks.	Reduction in development and commissioning time; productivity gains observed during validation.
	8.8	Improved correctness and predictability of control software enhance operational safety and reliability in industrial environments.	Reduction in software-induced faults; improved execution behavior during system validation.
12	12.2	By improving the correctness and efficiency of industrial control software, the proposed approach contributes to more efficient use of resources during production processes.	Reduction in production downtime; improved process efficiency enabled by reliable automation software.
	12.5	Increased software reliability and maintainability reduce waste associated with rework, debugging, and repeated commissioning activities in industrial automation projects.	Decrease in re-engineering effort; reduction in software-related production interruptions.
	12.6	Encourages sustainable software development practices by improving maintainability and life cycle efficiency.	Code complexity metrics; maintainability index; life cycle cost reduction.
4	4.4	The dissertation contributes to the development of advanced technical skills in automation engineering by bridging formal modeling techniques and industrial PLC implementation.	Availability of reusable methodologies, documentation, and toolchain prototypes for educational use.
	4.7	By promoting model-driven and standards-based engineering practices, the work supports education for sustainable and responsible industrial development.	Adoption potential in academic courses and professional training contexts.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Paulo José Lopes Machado Portugal, for their guidance, invaluable advice, and continuous support throughout this dissertation. Their expertise, constructive feedback, and encouragement were fundamental to the successful completion of this work.

I also wish to thank the faculty and staff of the Department of Electrical and Computer Engineering at the Faculty of Engineering of the University of Porto for providing an excellent academic environment and the resources necessary to carry out this research.

Finally, I am deeply grateful to my family for their unwavering support, patience, and belief in me throughout my studies. Their encouragement has been a constant source of motivation and has made this achievement possible.

To everyone who contributed, directly or indirectly, to this dissertation, I offer my thanks.

André Filipe Fernandes de Azevedo Osório

"I don't want to believe. I want to know."

Carl Sagan

Contents

Abstract i

UN Sustainable Development Goals ii

Acknowledgements iv

List of Figures x

List of Listings xi

List of Acronyms xii

1 Introduction 1
- 1.1 Problem . . . 1
- 1.2 Motivation . . . 1
- 1.3 Objectives and Possible Applications . . . 2
- 1.4 Document Structure . . . 2

2 Background 4
- 2.1 GRAFCET . . . 4
- 2.2 GRAFCET-Studio . . . 16
- 2.3 CODESYS and IEC 61131-3 . . . 22
- 2.4 PLCOpenXML . . . 25
- 2.5 OPC-UA . . . 29
- 2.6 Summary . . . 30

3 Related Work 31
- 3.1 Tools Researched . . . 31
  - 3.1.1 Tool for GRAFCET Creation . . . 31
  - 3.1.2 Translators for IEC 60848 to IEC 61131-3 . . . 33
  - 3.1.3 Target Tool for IEC 61131-3 Code . . . 35
  - 3.1.4 Other Tool Considerations . . . 35
- 3.2 GRAFCET Interpretation Algorithm Rules . . . 36
- 3.3 Meta Model for GRAFCET . . . 38
- 3.4 Target IEC 61131-3 Language . . . 41
- 3.5 Framing and Contribution to the SoA . . . 42

4 Toolchain Architecture **44**
4.1 Proposed Architecture . . . 44
4.2 GRAFCET Interpretation Algorithm . . . 45
4.3 Meta Model . . . 48
4.3.1 Objective . . . 49
4.3.2 Description . . . 49
4.4 PLC Program Structure . . . 52
4.5 Summary . . . 53

5 Toolchain Implementation **54**
5.1 Layout . . . 54
5.1.1 Directory Structure . . . 54
5.1.2 Usage Flow . . . 56
5.2 GRAFCET-Studio Reader . . . 56
5.2.1 Objectives . . . 56
5.2.2 Libraries . . . 57
5.2.3 Program Flow . . . 57
5.2.4 Outputs . . . 59
5.2.4.1 IntermediateSave.xml . . . 59
5.2.4.2 SymbolSave.xml . . . 62
5.3 Structured Text Translator . . . 63
5.3.1 Objectives . . . 63
5.3.2 Libraries . . . 63
5.3.3 Program Flow . . . 64
5.3.4 Outputs . . . 67
5.3.4.1 CODESYSProject.xml . . . 67
5.3.4.2 SymbolConfiguration.xml . . . 69
5.4 CODESYS Import Script . . . 70
5.4.1 Objectives . . . 70
5.4.2 Libraries and API . . . 70
5.4.3 Program Flow . . . 70
5.4.3.1 Information Gathering Phase . . . 71
5.4.3.2 Main Project Import Phase . . . 72
5.4.3.3 Symbol Configuration Import Phase . . . 72
5.5 Summary . . . 73

6 Project Validation **75**
6.1 Objectives . . . 75
6.2 Testing Process . . . 76
6.3 GRAFCET Tests . . . 77
6.3.1 Test 1 - Simple Elements . . . 77
6.3.2 Test 2 - Transition Types . . . 78
6.3.3 Test 3 - Action Types . . . 79
6.3.4 Test 4 - Macro Steps . . . 80
6.3.5 Test 5 - Enclosing Steps . . . 81
6.3.6 Test 6 - Forcing Orders . . . 82
6.3.7 Test 7 - Conditions . . . 84
6.3.8 Tests 8 and 9 - Realistic Grafcet Implementations . . . 86
6.4 OPC-UA Test . . . 89

6.5 Summary 92

7 Conclusion **93**
7.1 Understanding of the Problem 93
7.2 Results and Findings 94
7.3 Potential Contributions 95
7.4 Future Improvements 96

References **98**

A Toolchain User Manual **101**
A.1 Toolchain Installation 101
A.2 Toolchain Usage 103

B Toolchain File Structure Listings **106**
B.1 CODESYSProject.xml File 106
B.2 SymbolConfiguration.xml File 112

C JSON Model **114**

D GRAFCET Transformation Rules **117**

List of Figures

2.1 Example of simple GRAFCET diagram with key elements. 5
2.2 Example of an initial step and a normal step. 5
2.3 Example of a transition and its sub-types. 6
2.4 Example of a parallelization. 6
2.5 Example of a synchronization. 7
2.6 Example of continuous and stored actions. 7
2.7 Example of standalone horizontal and vertical directed links and example of their use in a chart. 8
2.8 Example of chart using a linked label. 8
2.9 Example of connected grafcets. 9
2.10 Example of a partial grafcet outline. 9
2.11 Example of a macro step and its associated partial grafcet. 10
2.12 Example of an initial and normal enclosing step and their associated partial grafcet. 10
2.13 Example of all forcing order types. 11
2.14 Example of a sequence structure. 11
2.15 Example of a sequence cycle structure. 12
2.16 Example of a selection structure. 12
2.17 Example of a forward skip structure. 13
2.18 Example of a backward skip structure. 14
2.19 Example of structure combining a synchronization and a parallelization. 14
2.20 Example of a chart that will always evolve in a non-transient way (left) and of a chart that will always evolve in a transient way (right). 16
2.21 Example of simple GRAFCET diagram with key elements in GRAFCET-Studio. 17
2.22 ST example code. 23
2.23 FBD example code. 23
2.24 SFC example code. 24
2.25 LD example code. 25
2.26 Basic workings of the OPC-UA protocol. 29

3.1 Meta model as presented by [14]. 40

4.1 Proposed architecture for toolchain. 45
4.2 Grafcet hierarchy problem example. 46
4.3 GRAFCET interpretation algorithm. 48
4.4 GRAFCET meta model. 50

5.1 Toolchain directory hierarchy. 55
5.2 Flowchart representing GRAFCET-Studio Reader's program flow. 58
5.3 Flowchart representing Structured Text Translators' program flow. 65

6.1 Grafcet test chart 1. 78
6.2 Grafcet test chart 2. 79
6.3 Grafcet test chart 3. 80
6.4 Grafcet test chart 4. 81
6.5 Grafcet test chart 5. 82
6.6 Grafcet test chart 6. 84
6.7 Grafcet test chart 7. 86
6.8 Grafcet test chart for the GRAFCET standard, annex B, version 1. 87
6.9 Grafcet test chart for the GRAFCET standard, annex B, version 2. 88
6.10 Symbol configuration for test 1. 89
6.11 Symbols associated to grafcet instance "fb_MainGrafcet". 90
6.12 UaExpert's address space with symbol configuration symbols. 91
6.13 UaExpert's symbols associated to grafcet instance "fb_MainGrafcet". 91
6.14 UaExpert's data access view. 91

A.1 GitHub page for the toolchain with the "releases" section highlighted. 101
A.2 Releases page. 102
A.3 Example directory of successful toolchain zip extraction. 102
A.4 Toolchain compilation script "toolchain_compile.bat". 103
A.5 Creation and saving of grafcet chart in "shared" directory. 104
A.6 Creation of empty CODESYS project. 104
A.7 Execution of the CODESYS Import Script. 105
A.8 Resulting CODESYS project after script has finished. 105

C.1 Diagram depicting first part of JSON structure. 115
C.2 Diagram depicting second part of JSON structure. 116

D.1 Activity diagram depicting component translation algorithm - Transitions and Steps. 118
D.2 Activity diagram depicting component translation algorithm - Actions. 120
D.3 Rule 11 rule set expansion. 121

List of Listings

2.1 GRAFCET-Studio general save file example 17
2.2 GRAFCET-Studio save file example - element 20
2.3 GRAFCET-Studio save file example - page 21
2.4 GRAFCET-Studio save file example - symbol 21
2.5 CODESYS specific PLCOpenXML save file structure 26
5.1 IntermediateSave.xml XML structure 60
5.2 SymbolSave.xml XML structure 62
5.3 CODESYS Import Script - Part 1 71
5.4 CODESYS Import Script - Part 2 72
5.5 CODESYS Import Script - Part 3 73
B.1 "CODESYSProject.xml" file's task structure 106
B.2 "CODESYSProject.xml" file's global variable list structure 107
B.3 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 1 107
B.4 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.1 108
B.5 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.2 108
B.6 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.3 108
B.7 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.4 109
B.8 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.5 109
B.9 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.6 110
B.10 "CODESYSProject.xml" file's "PLC_PRG" structure - Part 3 110
B.11 "CODESYSProject.xml" file's POU structure - Part 1 111
B.12 "CODESYSProject.xml" file's POU structure - Part 2 111
B.13 "CODESYSProject.xml" file's POU structure - Part 3 112
B.14 "SymbolConfiguration.xml" file's symbol structure 113

List of Acronyms

API	Application Programming Interface
CFC	Continuous Function Chart
DOM	Document Object Model
EMF	Eclipse Modeling Framework
FB	Function Blocks
FBD	Function Block Diagram
FSM	Finite State Machine
GAL	Guarded Action Language
GMF	Graphical Modeling Framework
GRAFCET	*Graphe Fonctionnel de Commande Étape-Transition*
GUI	Graphical User Interface
GVL	Global Variable List
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IL	Instruction List
IoT	Internet of Things
JSON	JavaScript Object Notation
LD	Ladder Diagram
LLM	Large Language Model
LoRA	Low-Rank Adaptation
M2T	Model-to-Text
MDE	Model-Driven Engineering
MDSE	Model-Driven Software Engineering
MQTT	Message Queuing Telemetry Transport
OOP	Object Oriented Programming
OS	Operating System
PLC	Programmable Logic Controller
POU	Program Organization Units
RAG	Retrieval-Augmented Generation
SFC	Sequential Function Chart
SLA	Stable Location Automaton
SoA	State of the Art
ST	Structured Text
UUID	Universally Unique Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Problem

In industrial automation, GRAFCET (*Graphe Fonctionnel de Commande Etapes/Transitions*) - IEC 60848 - is utilized to formally and graphically specify sequential control logic [12, 21].

However, Programmable Logic Controllers (PLCs) do not execute GRAFCET directly; instead, they rely on IEC 61131-3 programming languages.

This means that a control engineer must translate the original specification GRAFCET chart into IEC 61131-3 compliant code, taking into account semantic differences and other possible caveats such as a certain constructs not being a one-to-one mapping.

Furthermore, the lack of an automated and standardized transformation approaches from GRAFCET to IEC 61131-3 creates inconsistencies between the original specification and the implemented control logic [20].

1.2 Motivation

The motivation for this work arises from both industrial needs and research challenges in modern automation systems. As automation solutions become increasingly complex, there is a growing need for higher levels of abstraction and for model-driven engineering approaches that allow engineers to manage this complexity in a systematic and reliable way [37]. At the same time, automation engineers are under pressure to achieve shorter development cycles while still guaranteeing functional correctness and compliance with international standards [35, 37].

In this context, the manual translation of control specifications into PLC code significantly undermines the benefits offered by formal specification languages such as GRAFCET, as it introduces errors, inconsistencies, and additional development effort [20]. Moreover, the widespread adoption of concepts such as digital twins, modular automation, and interoperable industrial systems increases the need for strong consistency between high-level models and their executable implementations [37, 38]. Consequently, there is a clear demand for integrated solutions that

seamlessly connect specification, automatic code generation and communication mechanisms, particularly in PLC-based automation environments [37].

Automating the transformation from GRAFCET to IEC 61131-3 code can significantly reduce development time, improve reliability, and enable seamless integration with external systems, contributing to smarter and more connected industrial solutions [16, 20].

1.3 Objectives and Possible Applications

The main object of the work developed and described in this document will be to address these issues by creating and implementing a toolchain which translates GRAFCET models into IEC 61131-3 code.

This toolchain will extract grafcets created using GRAFCET-Studio from its save files, accommodate the extracted data into the meta model and, finally, translate the data into IEC 61131-3 compliant code, generating a final file in the PLCOpenXML format.

Overall objectives include:

- Formalize GRAFCET meta model, representing all the relevant elements and eventual auxiliary classes.
- Create and implement rules that govern the translation of the GRAFCET constructs into IEC 61131-3.
- Expose relevant variables via OPC-UA.

Considering this, the work done may have many applications. Among these are the rapid and reliable generation of PLC code from formal control specifications with seamless integration of high-level models with implementation, reducing manual intervention and ensuring consistency between simulated models and real PLC behavior. In education and training, it can support learning environments where students move from formal models to executable automation systems. In maintenance and re-engineering, where modification and validation of control logic are easier through model-level changes rather than low-level code edits.

1.4 Document Structure

Regarding it's structure, the following document will begin in chapter 2 by going over the definition of some the concepts and tools necessary for a proper understanding of the work developed.

Followed by this, in chapter 3, information regarding the tools researched that target every major development point is presented. Additionally, the description of the related works with an accompanying discussion and the contextualization of this dissertation's topic in the state of the art (SoA).

The next chapter - chapter 4 - begins by going over the proposed toolchain architecture, followed by the GRAFCET interpretation algorithm developed. Then it presents the objectives and

description of the created GRAFCET meta model and how it reflects the IEC 60848 standard. The chapter ends by presenting the overall PLC program structure to be utilized in the final implementation.

Chapter 5, which discusses the toolchain implementation, will begin by addressing its layout and folder structure, followed by describing the two core tools and their objectives and concluding with a script for directly importing the data into CODESYS.

Contained in chapter 6, are multiple tests and their according descriptions that demonstrate the toolchain's abilities, starting with tests that target certain components/structures, followed by larger tests with most constructs present and finishing with a depiction of the variables exposed by OPC-UA.

Finally, in the conclusion (chapter 7), are exposed a summary of the understanding of the problem, the current results and findings, the potential contributions of the work and future improvements to be made.

Chapter 2

Background

This chapter presents the essential theoretical and technological foundations required to understand the scope and methodology of the present work. To contextualize the effort of the research, the following sections detail the core components: the GRAFCET specification language and its modeling environment (GRAFCET-Studio), the target industrial automation platform (CODESYS) with its supported programming languages, the standardized project exchange format (PLCOpen-XML), and the communication protocol (OPC-UA) used for system integration and data exposure. Additionally, there is a summary present at the end of the chapter.

2.1 GRAFCET

GRAFCET (*Graphe Fonctionnel de Commande Etapes/Transitions*), standardized as IEC 60848 [12], is a graphical formalism specifically designed for the specification of the behavior of sequential control systems. Its intuitive, flowchart-like notation makes it a good tool for control engineers to model, design, and document the logic of industrial automation processes before implementation. At its core, GRAFCET describes system evolution through a sequence of states and the conditions that trigger transitions between them. Taking this into consideration, the following figure, figure 2.1, displays an example grafcet containing the most common elements.

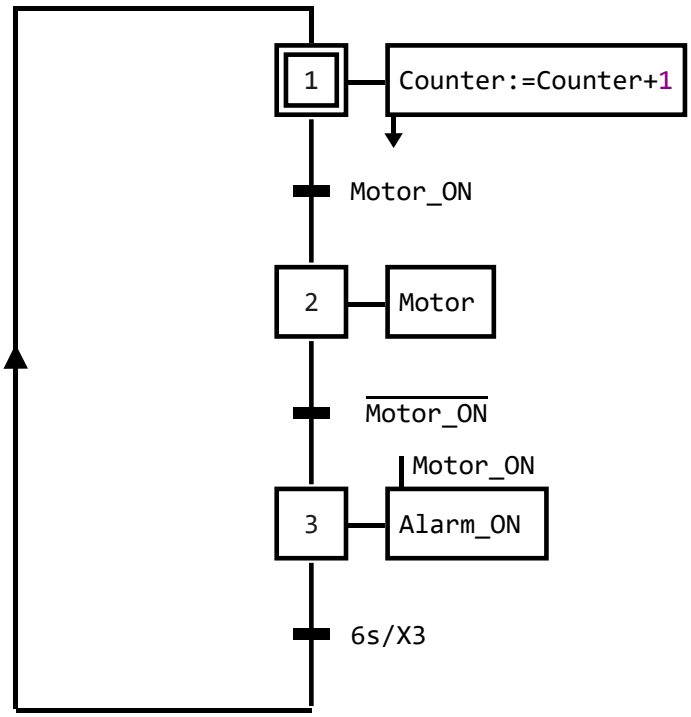


Figure 2.1: Example of simple GRAFCET diagram with key elements.

The fundamental building blocks of the GRAFCET specification are as follows:

- **Steps** - A step, depicted in figure 2.2, represents a stable state or phase of the controlled system. It is usually depicted by a square or a rectangle with a double line for an initial step (step activated at t=0s). They are referenced via a numerical depiction.

  A step can be either active or inactive, remaining as such until an outbound/inbound connected transition is cleared, at which point the step is deactivated/activated. At any given instance, the situation of the grafcet, its state, is represented by the set corresponding to the active steps of said grafcet chart.

  Additionally, a transition may be connected to one or multiple steps or vice-versa via constructs denominated directed links.

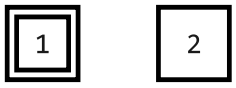


Figure 2.2: Example of an initial step and a normal step.

- **Transitions** - Transitions are the logical conditions that enable the system to progress from one set of active steps to the next. They are represented by a line perpendicular to the flow direction, as seen in figure 2.3.

  Transitions may be considered a normal transition, when connected to a previous and next elements, may be considered source transition, when there is only an option to connect to a next element and may further be a sink transition, when there is only the option of connecting to a previous element.

  Each transition is associated with a receptivity – a boolean expression - that must evaluate to true for the transition to "fire", also designated as to "clear". Said receptivity can be of multiple types: boolean expression, where the condition is based on a logic condition which can include mathematical expressions (e.g., Motor_ON, [Counter+2=5]); true/false, where the condition is always evaluated as true/false (e.g. 1, 0); rising or falling edge, where the condition is based on the state change of a boolean expression from false to true or from true to false respectively (e.g. ↑X1, ↓Motor_ON); time-dependent, where the expression "t1/a/t2" indicates that its value will replaced with true "t1" time units after "a" becomes true and shall remain true "t2" time units after "a" becomes false (e.g. 6s/X3). When a transition fires, it deactivates the preceding step(s) and activates the succeeding one(s).

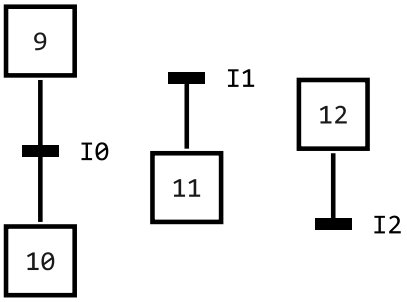


Figure 2.3: Example of a transition and its sub-types.

- **Parallelization** - Parallelizations are a type of transition represented by two horizontal lines, connected to transitions at the top and steps on the bottom - see figure 2.4. This element requires that only one of the previous transitions to be active for the grafcet state to advance, activating all the following steps.

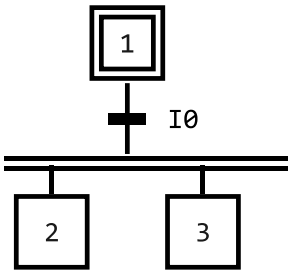


Figure 2.4: Example of a parallelization.

- **Synchronization** - Synchronizations are also a type of transition, akin to parallelizations, represented by two horizontal lines, connected to steps at the top and transitions on the bottom - see figure 2.5. This element requires all previous steps and any of the following transitions to be active for the grafcet state to advance.

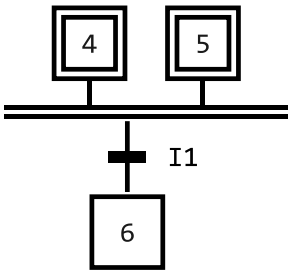


Figure 2.5: Example of a synchronization.

- **Actions** - Actions define the operations or commands executed while their associated step is active. They are each associated to a singular step and can be continuous actions (executed as long as the step is active - such as Motor in figure 2.1) which can be associated to conditions, where they will only be active so long as the condition is true (such as step 3's action in figure 2.1). Actions may also be of the stored variety, triggered once upon step activation, deactivation or due to an event and remain set until reset - as is the case for the Counter action. Figure 2.6 represents all action types.

  The conditions for the continuous actions and the events for the stored actions are much of the same type as those aforementioned for the transitions. The actions themselves are typically textual descriptions (e.g., Motor or Alarm_ON) placed beside the step in a rectangle of their own.

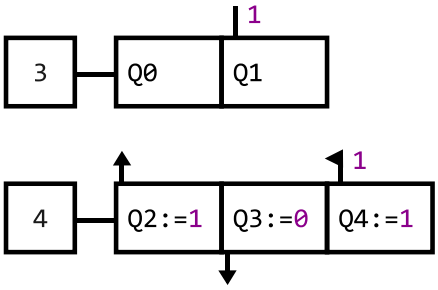


Figure 2.6: Example of continuous and stored actions.

- **Directed Links** - A directed link encompasses all the lines that connect steps, transitions, parallelizations, synchronizations and actions between one another, they may be vertical or horizontal. Figure 2.7 demonstrates an example of this component.

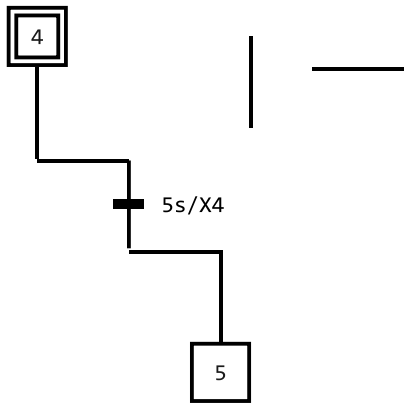


Figure 2.7: Example of standalone horizontal and vertical directed links and example of their use in a chart.

- **Linked Labels** - In the case that a directed link must be broken, a linked label may be used. It indicates a connection from a transition to a step, with the number over the linked label representing the step to which it will "jump" to, as shown in figure 2.8.

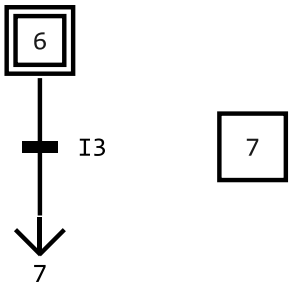


Figure 2.8: Example of chart using a linked label.

- **Connected Grafcets** - A connected grafcet is a structure in which a continuous path of links exists between any two or more elements. These component-component connections can be from step to transition, transition to step, step to action, step to synchronization, synchronization to transition, transition to parallelization or parallelization to step. Three connected grafcet examples are presented in figure 2.9. This element is not physical, representing instead a semantic association.

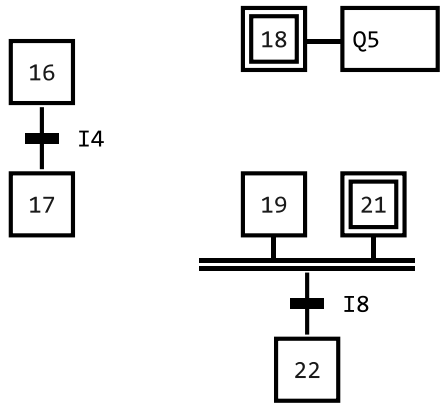


Figure 2.9: Example of connected grafcets.

- **Partial Grafcets** - These enable the decomposition of a large system into modular, interconnected units, composed by one or more connected grafcets. They are represented as an outline, as seen in figure 2.10, with two fields, one at the bottom for the name of the partial grafcet and the other at the top for the associated step if any (see Hierarchical Structures below). Partial grafcets can run concurrently and synchronize through common variables or transitions, facilitating the design of distributed or parallel processes.

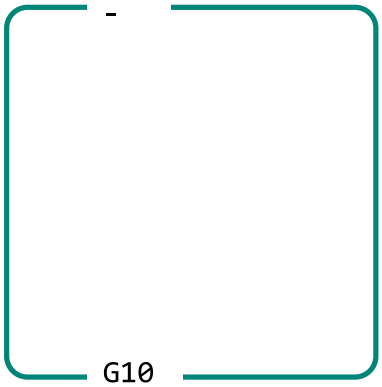


Figure 2.10: Example of a partial grafcet outline.

- **Hierarchical Structures** - To manage complexity, GRAFCET supports hierarchical modeling through two main constructs, which allows for abstraction and modularity, treating a complex subroutine as a single step within a higher-level diagram:

  - **Macro Steps** - A macro step, figure 2.11, is a step that encapsulates an entire subgrafcet sequence. This partial grafcet must contain two special steps, an entry step and an exit step. It must reach completion (exit step active) before the macro step can advance.

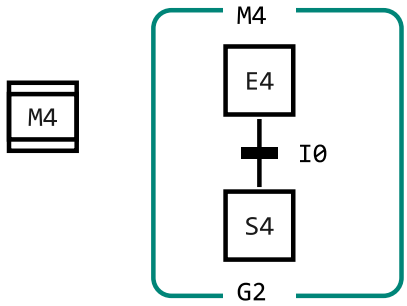


Figure 2.11: Example of a macro step and its associated partial grafcet.

- **Enclosing Steps** - An enclosing step, figure 2.12, is much the same as a macro step, differing only in the fact that it can encapsulate multiple partial grafcet sequences, that the enclosing step's following transition can 'fire' regardless of the sub-grafcet's progress and that it can be activated per the initial situation.

  Normal steps associated to a partial grafcet governed by an enclosing step may additionally have an asterisk to their left as a representation of the activation link. This link signals that, when the enclosing step, and subsequently the sub-grafcet, is activated, the steps associated with the activation link are to be enabled, unless said enclosing step activation pertains to the activation of the initial situation, where only the initial steps within the sub-grafcet are activated.

  Similarly to normal steps, enclosing steps can also be represented as belonging to the initial situation with the visual alteration of the addition of an outer square.

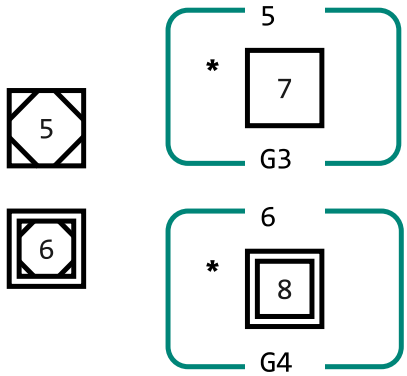


Figure 2.12: Example of an initial and normal enclosing step and their associated partial grafcet.

- **Forcing Orders** - Forcing orders provide a mechanism for overriding the normal evolution of the grafcet. They allow external events or conditions to directly force the activation of specific steps, activation of the initial situation (initial steps activate and all else deactivate) or deactivation all steps, regardless of the state of connected transitions. They can also force grafcets to be 'frozen', remaining in the state they are, irrespective of other alterations.

Figure 2.13 demonstrates all of these types. This is crucial for implementing emergency stops, manual overrides, or initialization sequences.



Figure 2.13: Example of all forcing order types.

GRAFCET is also composed of basic structures of interconnected elements, which may take the form of:

- **Sequence** - Multiple steps interconnected via transitions. The first step only has a singular following transition and no previous connections, whilst the last step only has a singular previous transition and no next element - see figure 2.14. Excluding these two steps, every other step is connected to a previous and following steps via two separate transitions.

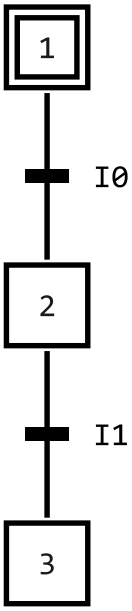


Figure 2.14: Example of a sequence structure.

- **Cycle of a Sequence** - Essentially the same as a sequence; however, the last step has a succeeding connection, via transition, to the first step of the sequence, see figure 2.15. This also implies that the first step has a preceding connection to the last step via the same transition. The flow of that particular transition's evolution goes from the last step to the first step in the sequence.

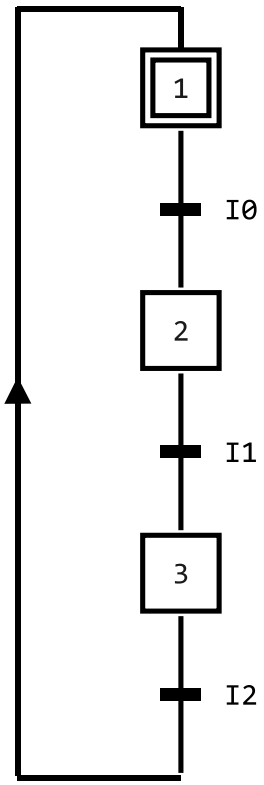


Figure 2.15: Example of a sequence cycle structure.

- **Selection** - A selection represents a choice of evolution between multiple possible sequences, where, usually, a singular step branches out with many transitions, see figure 2.16. The number of branching sequences has to be equal to the number of outgoing transitions.

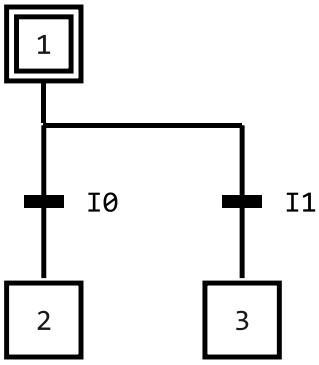


Figure 2.16: Example of a selection structure.

- **Forward Skip** - Forward skips are a type of selection where a particular sequence has two or more transitions that lead to it. For example, the first transition of a step leads to the

beginning of a sequence, while the other transition leads to the same sequence but one step ahead of the first, see figure 2.17.

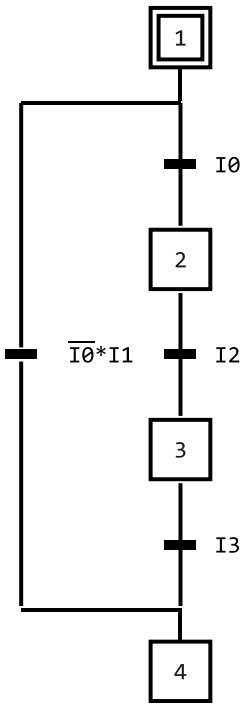


Figure 2.17: Example of a forward skip structure.

- **Backward Skip** - The backward skip is much the same as the forward skip, with the only notable difference being the fact that, instead of skipping to a step down the sequence, it return to a step upstream, see figure 2.18.

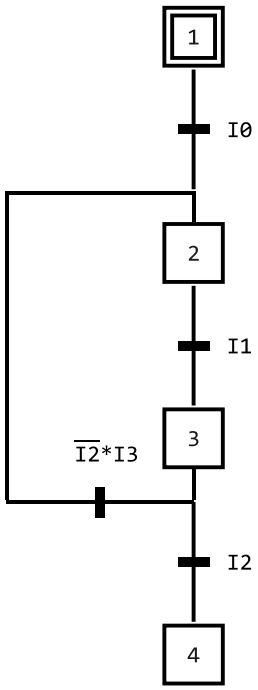


Figure 2.18: Example of a backward skip structure.

- **Synchronizations and Parallelizations** - These elements can be interpreted by themselves or as a combination of both. A standalone synchronization and parallelization were already described in previous paragraphs; however, they may both be used simultaneously in the same sequence, one after the other.

  This last particular case leads to a situation where all the steps previous to the synchronization must be active and the condition of the transition that connects both the synchronization and the parallelization must be evaluated to true in order for it to clear, activating all the steps ahead of the parallelization, thus acting as a synchronization for multiple branches of sequences with the added ability to easily convert any number of sequences to any other number of sequences. Figure 2.19 represents this structure.

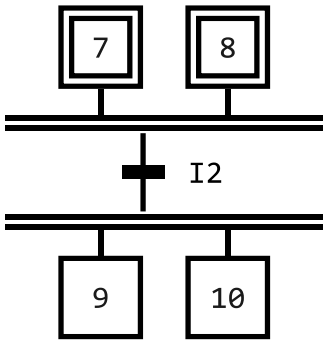


Figure 2.19: Example of structure combining a synchronization and a parallelization.

Furthermore, GRAFCET is defined by a set of five evolution rules that dictate it's functionality from state to state. These rules, as written in [12], are:

- **Rule 1**: The initial situation, chosen by the designer, is the situation at the initial time.
- **Rule 2**: A transition is said to be enabled when all immediately preceding steps linked to this transition are active. The clearing of a transition occurs:

  - when the transition is enabled, and

  - when its associated transition-condition is true.
- **Rule 3**: The clearing of a transition simultaneously provokes the activation of all the immediate succeeding steps and the deactivation of all the immediate preceding steps.
- **Rule 4**: Several transitions, which can be cleared simultaneously, are simultaneously cleared.
- **Rule 5**: If during the operation, an active step is simultaneously activated and deactivated, it remains active.

Moreover, the practical application of this set of evolution rules may lead a GRAFCET chart to evolve in one of two ways: non-transient evolution or transient evolution. Non-transient evolution denotes that an input event, this means the change of at least one input variable value, leads to the grafcet situation only advancing forward once. For example, when a transition is both active and its condition evaluates to true, the transitions clears, the previous step is deactivated and the following one is activated. On the other hand, transient evolution happens when, for example, there are three steps and two transitions that sequentially connect said steps. If the first transitions clears and the second one is already able to clear as well, the middle step is not activated as the state changes from the first step to the last, the middle step is considered to be activated and deactivated "virtually". This latter evolution type may have adverse effects on the functionality and dependability of the grafcet chart as it creates unstable situations. The figure below, figure 2.20, illustrates two situations regarding these types of evolution; one demonstrates a situation which will never lead to transient evolution (non-transient evolution) and the other will always lead to transient evolution.

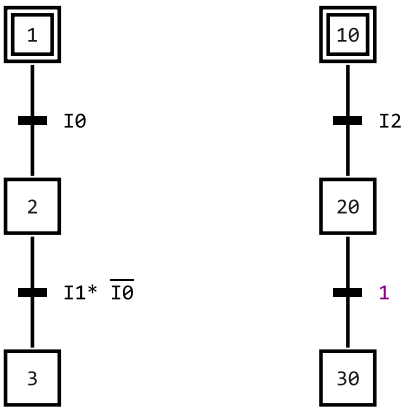


Figure 2.20: Example of a chart that will always evolve in a non-transient way (left) and of a chart that will always evolve in a transient way (right).

Owing to the possible ambiguities arising from not only this set of rules but also from the way the GRAFCET elements interact with each other and transient evolution, the IEC 60848 is considered by some to be a semi-formal standard, which is detrimental for the development of an accurate and consistent translation algorithm [21]. As such, a formal definition was necessary. Details regarding this definition can be found in the following chapter 3.

2.2 GRAFCET-Studio

GRAFCET-Studio [18] is a dedicated software tool for creating, simulating, and documenting GRAFCET diagrams in full compliance with the IEC 60848 standard. In this context, it plays the role of a source environment for the transformation toolchain.

It allows for a faithfully implementation of all core GRAFCET elements described in section 2.1 and, additionally, it permits the creation of "pages", which act as "main" grafcets - grafcets with the highest possible rank in the grafcet hierarchy - where all the remaining elements can be placed upon, with the possibility of inter-page interactions and checks.

GRAFCET-Studio contains a different naming convention, relative to GRAFCET, for certain elements, among them are the connection lines (directed links) and the jumps (linked labels), which from here on out will be used instead of the one in the standard.

Furthermore, this tool contains a "I/O-Panel" where variables can be manipulated and monitored whilst a simulation is ongoing.

Below is a figure of the GRAFCET example presented in figure 2.1. This video demonstrates a simulation of this grafcet.

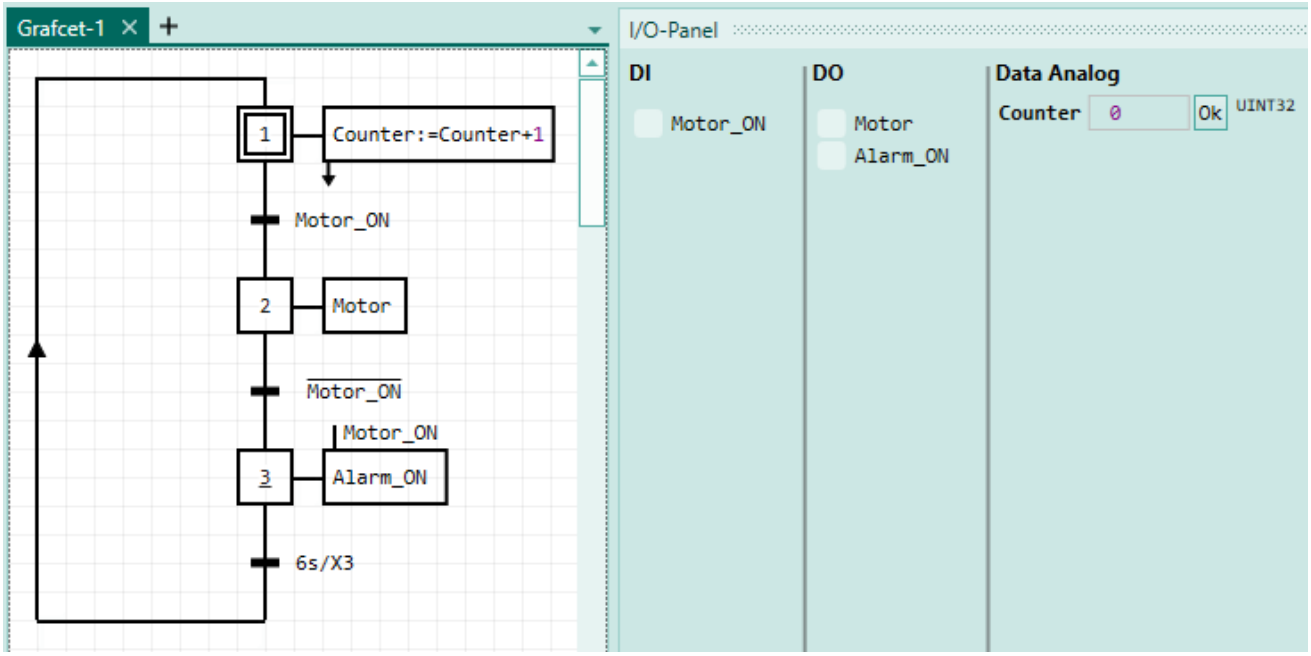


Figure 2.21: Example of simple GRAFCET diagram with key elements in GRAFCET-Studio.

The tool contains additional modules, namely, integration with "PLC-Lab", which functions as a moderately realistic process simulator with the ability to place objects and assign them particular properties and connections to the variables defined in GRAFCET-Studio. It proves itself to be irrelevant in the context of this dissertation as, although it allows for PLC integration, it does not allow for the inspection of said PLC integrated code, thus clashing against the desire for an open-source alternative of this work.

Another module is the "Grafcet-Engine", which allows GRAFCET-Studio to run its simulation on other platforms, be it a PLC, Arduino or even establishing a MQTT or OPC-UA connection. This last one allows for CODESYS integration; however, the given GRAFCET-Studio-to-CODESYS libraries are seemingly outdated, meaning that CODESYS must be rolled back to an earlier version, which proves undesirable as keeping the tools used up-to-date and still providing the desired functionality from the toolchain is part of this work's objectives.

With this in mind, the last feasible option for using this tool as a grafcet source comes from its save files. GRAFCET-Studio stores projects in a structured eXtensible Markup Language (XML) formatted file, demonstrated in listing 2.1. The file's root tag is "<GrafcetProjectFileSystemV1>", being subdivided into three main elements "<Grafcets>", which contains declarative data describing every grafcet and the elements in said charts, "<GrafcetsProperties>", which contains information regarding the pages of the project, and "<Symbols>", which contains the relevant variables and their descriptors. Moreover, it contains the tags "<SimulatorIoSettings>" and "<Settings>", which are not relevant nor were considered for this project.

Listing 2.1: GRAFCET-Studio general save file example

```
<?xml version="1.0" encoding="utf-16"?>
```

```
<GrafcetProjectFileSystemV1 xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
    XMLSchema">
  <Grafcets>
    <ArrayOfGrafcetItemProperties>
      * Element information goes here *
    </ArrayOfGrafcetItemProperties>
  </Grafcets>
  <GrafcetsProperties>
    * Page information goes here *
  </GrafcetsProperties>
  <Symbols>
    * Symbol information goes here *
  </Symbols>
  <SimulatorIoSettings>
    * SimulatorIoSettings information goes here *
  </SimulatorIoSettings>
  <Settings>
    * Settings information goes here *
  </Settings>
</GrafcetProjectFileSystemV1>
```

Regarding the following listings (listings 2.2 to 2.4), which represent snippets of the XML save file from the example in figure 2.21, note that XML elements considered unnecessary to the understanding of the file structure were remove and that the expression "(...)" is used to convey that there are more similar elements belonging to a given tag, but were omitted to much the same reason as the previous XML elements.

Starting with the description of listing 2.2: it begins with the tag "<Grafcets>", representing the list of all present grafcets which are denoted by the tag "<ArrayOfGrafcetItemProperties>". This latter tag is then subdivided further into each element of said grafcet, represented by "<GrafcetItemProperties>", which itself contains a multitude of tags representing attributes of a certain element. These tags are as follows:

- **<IsHorizontalTransition>** - Denotes whether an element of <TypeEnum> of value Transition is horizontal (true) or not (false). Default value is "false".
- **<IsStartStep>** - Refers to whether a step has an activation link (true) or not (false). Default value is "false".
- **<Margin>**

  - **<Left>**, **<Top>**, **<Right>**, **<Bottom>** - These tags refer to the position of an element, usually assigned by the top-left most vertex of a given element. "Left" refers to its "X"

value (left to right), "Top" to its "Y" value (top to bottom), whilst "Right" and "Bottom" have always a null value, therefore these last two were not considered in the context of this dissertation.

- **<Height>** and **<Width>** - Refer to the height and width, respectively, of the gray box that envelops the element when selected in the editor. Sometimes these values do not accurately represent the actual visual dimensions of the elements themselves.
- **<TypeEnum>** - Refers to the type of element being analyzed. The value of this tag reflects, for the most part, the element in question, for example, "<TypeEnum>Transition</TypeEnum>" refers to a transition. There are however three exceptions, which are the partial grafcet borders, synchronizations and parallelizations, referred to as "GroupBorder", "ParallelConnectionBottom" and "ParallelConnectionTop" respectively. Below is a list of equivalence between the norms' constructs and their TypeEnum in GRAFCET-Studio:
  - Normal Step - StepNormal
  - Normal Initial Step - StepInitial
  - Macro Step - StepMakro
  - Enclosing Step - StepEnclosing
  - Enclosing Initial Step - StepInitialEnclosing
  - Transition - Transition
  - Continuous Action - ActionNormal
  - Continuous Action with Condition - ActionConditional
  - Stored Action on Activation - ActionActivation
  - Stored Action on Deactivation - ActionDeactivation
  - Stored Action on Event - ActionEvent
  - Forcing Order - ActionForcing
  - Partial Grafcet Border - GroupBorder
  - Synchronization - ParallelConnectionBottom
  - Parallelization - ParallelConnectionTop
- **<ConditionStr>** - Refers to the boolean conditions that both the continuous action with condition and the stored action on event must clear in order for their action to be executed. Default value is "1".
- **<TransitionStr>** - Exclusive to transitions and refers to their boolean condition that must become true so that they may clear. Default value is "".
- **<ActionStr>** - Refers to the text of the action of all action elements including forcing orders. Default value is "?".
- **<Caption>** - Captions has different meanings for different elements. When referring to a step, the value indicates the name given to said step. When referring to a group border, it denotes the macro or enclosing step caption of the calling step in the case that the partial grafcet is assigned to one. Finally, when referring to a jump, it simply indicates the following step the connection shall be made to. Default value is "".

- **<IsHorizontalLine>** - Exclusive to connection lines, refers to whether the connection line is horizontal (true) or vertical (false). Default value is "false".
- **<IsDirectionToTop>** - Indicates whether the direction of the logic flow of a horizontal transition (<IsHorizontalTransition>true</IsHorizontalTransition>) is from an element that is lower to an element that is higher (higher Y to lower Y) (true) or vice-versa (false). An example of this tag being true is the aforementioned figure 2.18 of section 2.1. Default value is "false".
- **<GroupName>** - Refers to the name given to a group border, representing the name of a partial grafcet. Default value is "-".
- **<IsSourceTransition>** - Valued at "true" if a given transition is a source transition, otherwise it is "false". Default value is "false".
- **<IsFinalTransition>** - Similar to "<IsSourceTransition>" but for sink transitions, "true" if sink transition, "false" otherwise. Default value is "false".

Listing 2.2: GRAFCET-Studio save file example - element

```
<Grafcets>
  <ArrayOfGrafcetItemProperties>
    <GrafcetItemProperties>
      <IsHorizontalTransition>false</IsHorizontalTransition>
      <IsStartStep>false</IsStartStep>
      <Margin>
        <Left>128</Left>
        <Top>32</Top>
        <Right>0</Right>
        <Bottom>0</Bottom>
      </Margin>
      <Height>32</Height>
      <Width>32</Width>
      <TypeEnum>StepInitial</TypeEnum>
      <ConditionStr>1</ConditionStr>
      <TransitionStr />
      <ActionStr>?</ActionStr>
      <Caption>1</Caption>
      <IsHorizontalLine>false</IsHorizontalLine>
      <IsDirectionToTop>false</IsDirectionToTop>
      <GroupName>-</GroupName>
      <IsSourceTransition>false</IsSourceTransition>
```

```
        <IsFinalTransition>false</IsFinalTransition>
      </GrafcetItemProperties>
      (...)
  </ArrayOfGrafcetItemProperties>
</Grafcets>
```

The second listing, listing 2.3, describe the page of the given GRAFCET-Studio project. The tag "<GrafcetsProperties>" refers to the list of pages, while "<GrafcetItemCollectionHelper>" refers to the individual pages themselves, each of them containing only one tag, which is "<Title>", referring to name attributed to a certain page.

Listing 2.3: GRAFCET-Studio save file example - page

```
<GrafcetsProperties>
  <GrafcetItemCollectionHelper>
    <Title>Grafcet-1</Title>
  </GrafcetItemCollectionHelper>
</GrafcetsProperties>
```

Lastly, listing 2.4 goes over the save file structure used for a particular symbol from the overall variable list from the project, in this example the symbol is "Motor_ON". The symbol section of the save file begins with the tag "<Symbols>", which represents the list of variable tag items. The variables themselves are described by "<SymbolItem>", containing three tags: "<Symbol>" denotes the name of the variable, "<DataType>" denotes the type of data, which can be BOOL, INT16, INT32, UINT8, UINT16, UINT32 or FLOAT and finally "<SymbolType>" denotes the type of the variable itself, which can be Input, Output or Data.

Listing 2.4: GRAFCET-Studio save file example - symbol

```
<Symbols>
  <SymbolItem>
    <Symbol>Motor_ON</Symbol>
    <DataType>BOOL</DataType>
    <SymbolType>Input</SymbolType>
  </SymbolItem>
  (...)
</Symbols>
```

However, a significant challenge arises from the fact that GRAFCET-Studio operates as a "black box" regarding the semantic relationships between elements. The XML file lists elements

and their properties but does not explicitly encode the logical connections (e.g., which transition follows which step). These connections are, as far as the end user is concerned, inferred internally by the software based on the graphical layout. Therefore, without any other indication of what may the element association method be, any external tool must reconstruct these semantic links by analyzing the spatial coordinates and dimensions of the elements — a process of coordinate-based interpretation, discussed in chapter 5, section 5.2.

2.3 CODESYS and IEC 61131-3

CODESYS [4] is a powerful, hardware-independent integrated development environment (IDE) and runtime system for building PLC code. At its core is the implementation of the IEC 61131-3 international standard [13], which defines a suite of programming languages for industrial automation. The transformation toolchain developed targets CODESYS as the deployment platform.

PLC's are beholden to a cycle of execution, whereby it begins with reading the input variables, followed by the execution of the logic program itself and finishing with the writing of the output variables. These task cycles have multiple types and, although CODESYS implements more, the main two are cyclic, where as soon as the outputs are written the inputs begin being read, and periodic, where the whole cycle of read, execute and write is inserted into equally timed slots, executing another iteration only after a certain interval of time units have elapsed.

The tasks may also have differing priorities, where the higher priority task takes precedence over others with lower priorities. With preemptive scheduling a lower priority task is interrupted so that the higher priority task may execute, the former only continuing execution when all other higher priority task are done. On the other hand, non-preemptive scheduling forces the higher priority task to await the lower priority task's cycle to finish.

The logic programs executed by PLC's are referred to as Program Organization Units (POUs). As defined by the IEC 61131-3 standard, these fall into three types.

Program and function block POUs are similar in that both can have multiple input and output parameters and can retain data between executions. The key distinction between them is that function blocks must be instantiated before they can be used. Each instance maintains its own memory, allowing multiple independent instances of the same function block to operate simultaneously. In contrast, programs cannot be instantiated and are executed only by tasks defined in the PLC configuration.

A function POU can accept multiple input parameters but produces a single output value and does not retain data between executions. It is designed to return the same result whenever it receives the same inputs. Functions have no internal memory, cannot access global variables, and may only invoke other functions.

Furthermore, the IEC 61131-3 standard defines five languages, combining graphical and textual paradigms:

- **Structured Text (ST)** - A high-level, Pascal-like textual language. It is ideal for complex algorithms, mathematical computations, data handling, and iterative loops. Its structured

nature makes it highly suitable for the automated generation of code from formal models, which is why it is the target language for this dissertation's transformation toolchain, more information in chapter 3. Figure 2.22 shows an example.

```
1   PROGRAM POU
    VAR_INPUT
3       S1 : BOOL ;
4       S2 : BOOL ;
5   END_VAR
6   VAR
7       A1 : BOOL := FALSE ;
8       A2 : BOOL := FALSE ;
9   END_VAR
```

```
1   IF S1 AND S2 THEN
2       A1 := TRUE ;
3       A2 := TRUE ;
4   ELSE
5       A1 := FALSE ;
6       A2 := FALSE ;
7   END_IF
```

Figure 2.22: ST example code.

- **Function Block Diagram (FBD)** - A graphical language where the program is constructed by interconnecting function blocks (representing functions or function blocks). Data flows from the outputs of one block to the inputs of another - figure 2.23. It is particularly well-suited for continuous process control and signal processing.

```
1   PROGRAM POU
2   VAR_INPUT
3       S1 : BOOL ;
4       S2 : BOOL ;
5   END_VAR
6   VAR
7       A1 : BOOL := FALSE ;
8       A2 : BOOL := FALSE ;
9   END_VAR
```

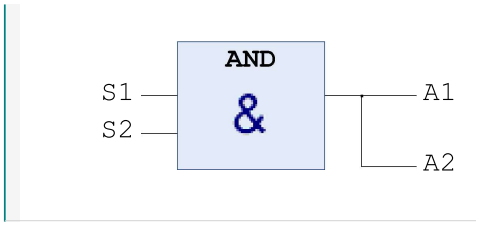


Figure 2.23: FBD example code.

- **Sequential Function Chart (SFC)** - A graphical, state-based language that bears a strong resemblance to GRAFCET itself. It organizes logic into steps, transitions, and actions. While seemingly the most direct mapping target for GRAFCET, its implementation and execution semantics in PLCs can differ from the specification semantics of GRAFCET.

```
1   PROGRAM POU
2   VAR_INPUT
3       S1 : BOOL ;
4       S2 : BOOL ;
5   END_VAR
6   VAR
7       A1 : BOOL := FALSE ;
8       A2 : BOOL := FALSE ;
9   END_VAR
```

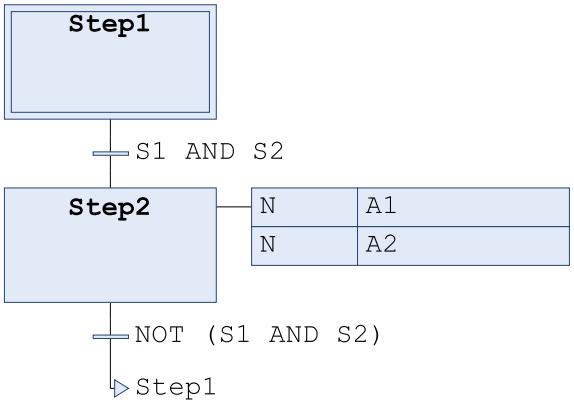


Figure 2.24: SFC example code.

- **Ladder Diagram (LD)** - Originally designed to mimic electrical relay logic diagrams for ease of adoption by electricians. It is based on the concept of power rails and rungs with contacts and coils. A newer and easier version of LD, also referred to as Ladder Diagram (LD2), was added to CODESYS, but for all intents and purposes they function much the same.

```
1       PROGRAM  POU
2       VAR_INPUT
3           S1 :  BOOL ;
4           S2 :  BOOL ;
5       END_VAR
6       VAR
7           A1 :  BOOL  :=  FALSE ;
8           A2 :  BOOL  :=  FALSE ;
9       END_VAR
```

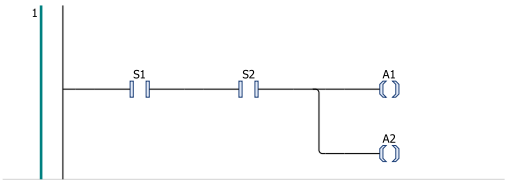


Figure 2.25: LD example code.

- **Instruction List (IL)** - A low-level, assembly-like textual language. It has been deprecated in the third edition of the IEC 61131-3 standard and is not relevant for modern code generation or this dissertation.

CODESYS provides an additional language, **Continuous Function Chart (CFC)** and its page-oriented variant. CFC extends the concept of the standard FBD by removing many of the layout restrictions found in it. Instead of arranging logic into discrete networks with predefined execution flow, CFC allows for the placement of function blocks anywhere on a canvas and wire them together naturally. However, it proves to be irrelevant in the context of this dissertation as it does not belong to the IEC 61131-3 standard, which is the transformation objective.

2.4 PLCOpenXML

To facilitate the transfer of automation application projects in IEC 61131-3 between different engineering tools, PLCOpenXML [30] was developed as a vendor-neutral, XML-based exchange format. It is maintained by the PLCOpen organization [29] and standardized as part of IEC 61131-10. It allows for greater portability, where the transfer of project files between tools is facilitated, automation, where integration with overall workflows, validation chains, script (such as the one developed in this dissertation's work) and other software is easier and, finally, version control, allowing for simpler tracking than if binary files were used.

For this project, PLCOpenXML serves as the crucial integration layer between the tool generating the IEC 61131-3 code and CODESYS. The transformation toolchain does not need to generate CODESYS-specific proprietary project files. Instead, it generates an IEC 61131-3 compliant program structure (POUs, data types, variables, tasks, configurations, metadata) and serializes it into a PLCOpenXML file. This standardized file can then be directly imported into the CODESYS development environment or other compatible tools, where it is recognized as a valid

project component. This approach ensures toolchain independence from CODESYS's internal application programming interfaces (APIs) and focuses the effort on the semantic translation from GRAFCET to IEC 61131-3 constructs.

CODESYS supports PLCOpenXML formatted files; however, the project structure of a PLCOpenXML file from CODESYS has some minor additions relative to the IEC 61131-10 standard. Since CODESYS is the target tool of the toolchain developed, the CODESYS PLCOpenXML structure implementation is the one being considered in this work - see listing 2.5.

It begins with the root tag "<project>", which expands to five tags.

Firstly, is the "<fileHeader>", which contains the attributes "companyName", "productName", "productVersion" and the "creationDateTime", with the product in question being CODESYS.

Secondly, is the "<contentHeader>", addressing the attributes "name" and "modificationDateTime" relating to the project. This tag contains the additional tag "<coordinateInfo>", that addressed scaling information for the graphical languages (FBD, LD and SFC) and the tag "<addData>" with extra project information.

Thirdly, is the "<types>", which is irrelevant for this dissertation as it goes seemingly unused by CODESYS, at least for the required application. Regardless, it contains empty tags for "<dataTypes>" and "<pous>".

Fourthly, is the "<instances>" tag, which expands into a "<configuration>" tag, which itself expands into a "<resource>" and "<addData>" tags. The "<addData>" tag contains information regarding the configuration of the device. On the other hand, the "<resource>" tag details the tasks ("<task>"), the global variable lists ("<globalVars>") and the POU data ("<pou>"). The task tag addresses task related information such as the task name, task priority, the POU instances being considered ("<pouInstance>"), the type of task cycles, the time interval of the cycles and the watchdog information. The global variables tag lists all the variables contained in it by using multiple "<variable>" tags. These variable tags contain a name attribute and a "<type>" tag which can take many values, the ones relevant to this work are: derived (when the variable is an instance of a function block POU), BOOL, INT, DINT, USINT, UINT, UDINT and REAL. The variables may additionally, if provided with one, contain an initial value ("<initialValue>"). The final tag is related to the POUs, wrapped in a "<addData>" and "<data>" tags, containing the name and POU type attributes. Additionally, it contains an "<interface>" tag, that, just like the global variable list tag, contains all the relevant variable elements, and the <"body"> tag, which details the type of language used in the POU with an appropriate tag (e.g. "<ST>") and the code itself within this previous and a "<xhtml>" tags. After this, there can be multiple additional "<data>" tags belonging to the initial "<addData>" tag, which reference the methods belonging to the POU if any. These methods contain an equal structure to that of the aforementioned POU tag.

Lastly, a simple "<addData>" tag that addresses the project structure hierarchy - devices, applications, tasks, global variable lists, POUs and methods, etc.

Listing 2.5: CODESYS specific PLCOpenXML save file structure

```
<project>
```

```
<fileHeader companyName="" productName="CODESYS"
    productVersion="CODESYS␣V3.5␣SP21␣Patch␣3" creationDateTime
    ="2026-06-21T20:09:18.1176756" />
<contentHeader name="Untitled1.project" modificationDateTime="
    2026-06-21T20:08:32.8110794">
  <coordinateInfo>
    (1) Scaling information for FBD, LD and SFC
  </coordinateInfo>
  <addData>
    (2) "Project␣information"
  </addData>
</contentHeader>
<types>
  <dataTypes />
  <pous />
</types>
<instances>
  <configurations>
    <configuration name="Device">
      <resource name="Application">
        <task name="GrafcetTask" priority="1">
          <pouInstance name="PLC_PRG" typeName="">
            <documentation>
              <xhtml xmlns="http://www.w3.org/1999/xhtml" />
            </documentation>
          </pouInstance>
          <addData>
            (3) Task information (type time interval in <
               TaskSettings>, and watchdog in <Watchdog>
               inside the previous tag)
          </addData>
        </task>
        <globalVars name="GVL">
          <variable name="fb_MainGrafcet">
            <type>
              (4) Var types: derived and all the others
            </type>
          </variable>
        </globalVars>
        <addData>
```

```
            <data name="http://www.3s-software.com/plcopenxml/
                pou" handleUnknown="implementation">
              <pou name="PLC_PRG" pouType="program">
                <interface>
                  <localVars>
                    <variable name="globalStableState">
                      <type>
                        (5) Var types: derived and all the
                            others
                      </type>
                      <initialValue>
                        (6) Initial value
                      </initialValue>
                    </variable>
                  </localVars>
                </interface>
                <body>
                  <ST> type of language
                    <xhtml>
                      (7) The code itself
                    </xhtml>
                  </ST>
                </body>
              </pou>
            </data>
            <data>
              (8) Methods - similar to the pou itself
            </data>
            <data>
              (9) Libraries
            </data>
          </addData>
        </resource>
        <addData>
          (10) Device configuration
        </addData>
      </configuration>
    </configurations>
  </instances>
  <addData>
```

```
    (11) Data regarding the project structure (Device,
        Application, Tasks, Global Variable Lists, POUs and
        methods, etc)
  </addData>
</project>
```

2.5 OPC-UA

Open Platform Communications Unified Architecture (OPC-UA) [27] is a platform-independent communication standard designed for secure and reliable data exchange in industrial automation and control systems. It enables interoperability between devices, machines, and software applications from different vendors while providing built-in mechanisms for security, scalability, and information modeling.

Unlike traditional industrial communication protocols that primarily exchange raw process values, OPC-UA represents information through a structured and extensible information model. Variables, objects, methods, and their relationships can be described within a standardized address space, enabling clients to interpret not only the value itself but also its semantic meaning. OPC-UA follows a service-oriented architecture in which clients access the server address space through standardized services, including node browsing, variable reading and writing, method invocation, and subscriptions to data changes.

The following figure (figure 2.26) illustrates the basic OPC-UA client-server architecture. Data and associated metadata from one or more devices are exposed through an OPC-UA server and can subsequently be accessed by connected clients. Depending on the application requirements, information exchange may be configured as read-only, write-only, or read/write.

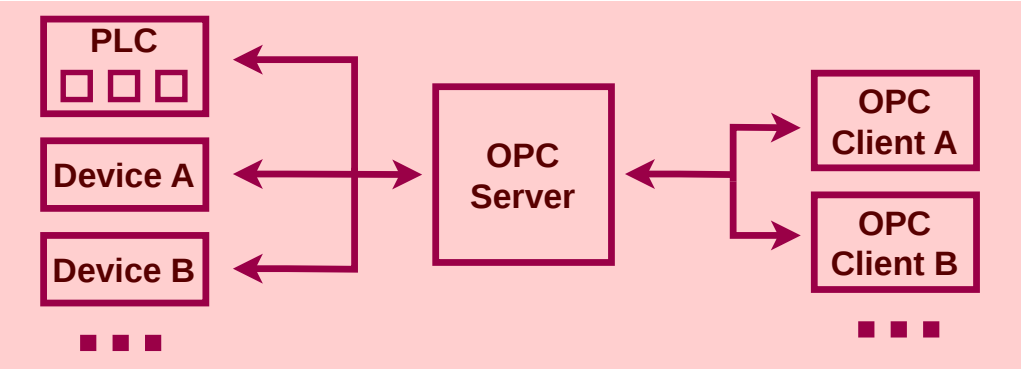


Figure 2.26: Basic workings of the OPC-UA protocol.

In the context of this dissertation, OPC-UA is the designated mechanism for runtime data exposure. One of the toolchain's objectives is to not only generate the control logic but also to instrument the code to publish relevant runtime information. This includes the active/inactive

status of GRAFCET steps, the current values of key variables, and the activation of transitions and actions.

Since CODESYS, the target tool of the generated IEC 61131-3 code, already has OPC-UA server capabilities integrated, the use of external tools will not required, with the exception of an OPC client.

2.6 Summary

This chapter has presented the fundamental knowledge regarding the main tools and technologies required for and adequate understanding of the work presented in this dissertation.

Firstly came the IEC 60848 norm, commonly referred to as GRAFCET, a graphical norm appropriate to describe the behavior of sequential control systems, followed by the description of all the graphical elements and their interactions. The core GRAFCET structures were also addressed, as well as the evolution rules and the the two types of evolution, transient and non-transient.

Followed by this norm, is the selected tool that allows for the creation of grafcets, GRAFCET-Studio. It begins by addressing the complementary graphical elements added by GRAFCET-Studio - jumps and connection lines. Followed by and image depicting the previous GRAFCET example in GRAFCET-Studio with an appropriate video of its simulation. After a further description of the modules , a full description of all the relevant XML tags of GRAFCET-Studio's XML save file is given, accompanied by relevant examples.

The second tool used is CODESYS, whose section describes how the IEC 61131-3 norm is incorporated into it. Not only are the main types of cycles addressed (cyclic and periodic), so are the types of scheduling (preemptive and non-preemptive) and the types of POUs (program, function block and function). Additionally, a light description of all the IEC 61131-3 languages and one exclusive to CODESYS are given, with appropriate examples.

Then comes PLCOpenXML (IEC 61131-10), more specifically the version of PLCOpenXML adapted to CODESYS. Apart from and overall description of the norm, a detailed analysis of the five main tags used in the CODESYS PLCOpenXML implementation is given.

Finally, OPC-UA, the communication protocol used for exposing the necessary variables from CODESYS, is addressed. It starts with a description of this protocol and how it differs from others, as it exposes the variables and the relations between each other. The section then describes the server-client communication and some of its configuration aspects with a visual aid. It finishes with a contextualization of OPC-UA relative to CODESYS and the project developed.

Having presented the fundamental tools, technologies, and standards that underpin this dissertation, the following chapter reviews the relevant State of the Art (SoA). It examines existing research, alternative tools and approaches, and related academic work.

Chapter 3

Related Work

This chapter presents a review of the current State of the Art (SoA) relevant to the topics addressed in this dissertation. It examines the existing research, methodologies, tools, and technologies that underpin the proposed work, providing the necessary context and rationale for the choices made throughout the dissertation. In addition to summarizing the contributions of previous studies, this chapter critically discusses their strengths and limitations, highlighting their relevance to the research objectives. Finally, it positions the present work within the existing body of knowledge, identifying the gaps it aims to address and outlining its expected contributions to the SoA.

3.1 Tools Researched

This section will address the tools researched to fulfill certain objectives. First, tools that are capable of creating charts according to the GRAFCET standard are shown. Secondly, tools that can translate IEC 60848 charts into to IEC 61131-3 code are presented, followed by a subsection that explains why CODESYS will be the target tool for said IEC 61131-3 code. The following subsection mentions the reasons behind choosing the communication protocol OPC-UA over MQTT. Lastly, some tools, which were found via the works researched, are addressed.

3.1.1 Tool for GRAFCET Creation

In order to carry out the envisioned work, a multitude of tools were analyzed to conclude their usefulness towards the set goals. This research began with tools that allow for the creation of GRAFCETs:

- **FluidSIM** - Developed by Festo [9], FluidSIM allows for designing and simulating grafcets according to the norm, as well as pneumatic, hydraulic, and electrical control systems. Within FluidSIM, GRAFCET can serve as a high-level control layer that can be directly linked to simulated sensors, actuators, and electrical components, allowing users to model, visualize, and test grafcet sequences. However, it does not allow for direct PLC control, meaning a separate implementation must be developed.

- **OFT2** - Developed by Omegon Teachware [26], OFT2 is a simulation and training environment for pneumatic, hydraulic, electrical, and sequential control systems. It allows for creation and simulation of grafcet sequences, where inputs and outputs can be analyzed via other means, such as through a USB I/O adapter connected to physical hardware.

- **LabVIEW** - Developed by National Instruments [23], LabVIEW is a general-purpose graphical programming environment, it allows for the creation of more state machine oriented programs although its focus is, predominantly, on data acquisition, display and general monitoring of application data. The tool does not natively support GRAFCET; however, there is an add-on, Grafcet Designer [7], that allows design and execution of grafcets, where the active steps of the model can visualized to debug and validate a given application.

- **WinErs** - Developed by Dr .- Ing. Schoop GmbH [11], WinErs is an automation and process control environment that integrates SCADA, HMI, data acquisition and an executable GRAFCET editor compliant with the IEC 60848 standard. It allows users to design, simulate, and execute GRAFCET sequences directly through its integrated soft PLC, supporting online debugging and interaction with both simulated and physical I/O devices.

- **Automation Studio** - Developed by Famic Technologies Inc. [8], Automation Studio is an engineering environment for the design, simulation, and documentation of hydraulic, pneumatic, electrical, and automation systems. It includes dedicated editors for both IEC 60848 GRAFCET and IEC 61131-3 Sequential Function Charts (SFC), allowing users to create and simulate sequential control applications while directly linking them to virtual actuators, sensors, and PLC logic. Additionally, it supports exporting GRAFCET and SFC models to formats compatible with Siemens S7 PLCs and XML.

- **SFCEDIT** - Developed by Stéphane Di Meglio [34], SFCEDIT is a graphical editor for developing GRAFCET (IEC 60848) models. It enables the creation of sequential control system specifications using the standardized graphical notation and provides methods for exporting the resulting diagrams in XML and graphical formats. As of writing, the tool has been deprecated and has little online presence.

- **Dia** - Developed by Alexander Larsson [1], Dia is an open-source diagramming application used to create technical diagrams, including flowcharts, UML diagrams, network topologies, and engineering schematics. It provides a library of predefined graphical elements and supports exporting diagrams to multiple vector and raster formats.

Taking this into consideration, a tool was selected. FluidSIM was not chosen as it does not allow for exporting grafcets in a easily readable format by other tools, it uses its native .fls format and image printing.

OFT2, although interesting for possible future applications owing to its ability to control physical hardware, was discarded in favor of another tool, as it also does not provide simple access to grafcet information.

LabVIEW focuses not on development of GRAFCET but its monitoring, it was discarded as a possibility for this work.

WinErs was not chosen for this work as it uses a proprietary project format and does not provide a straightforward mechanism for exporting the models into an representation suitable for external processing.

The lack of information regarding Automation Studio's save file and its XML export, lead it to being discarded as a possible tool as well.

Not only is SFCEDIT deprecated, but very little information regarding it is available, as such it was not considered.

Dia, being a simple digital diagram graphical editor, does not add much in the way of GRAFCET-specific capabilities, being equally discarded.

Overall, the program chosen for this particular part of the work was, ultimately, GRAFCET-Studio [18]. Not only does it provide itself in a simpler format, with easy access to all required components, simulation capabilities and monitoring, it also allows for easy acquisition of the grafcet structures via its save file, as outlined in section 2.2 of chapter 2. Apart from this, not only have I already used the tool on many occasions, with this experience proving useful, as it is already provided by the university, meaning that acquisition is a non-issue.

3.1.2 Translators for IEC 60848 to IEC 61131-3

Another key objective is to transform a grafcet into 61131-3 code. None of the previously mentioned tools allow for a direct translation to any of the IEC 611131-3 languages, at most they allow running the control logic by establishing communication with a given PLC. However, these following tools/scripts translate a given grafcet model into PLC code, they are:

- **grafcet2plc** - Created by Florentin Delaine [10], grafcet2plc generates PLC code from grafcets, making use of Python. The purpose of the software as described by its author is to provide an interface between programs that create grafcets and PLCs so that programming is faster.

- **G4LTL-ST** - Created by Chih-Hong Cheng, Chung-Hao Huang, Harald Ruess and Stefan Stattelmann [3], G4LTL-ST creates PLC code - specifically in the ST language - from "timed behavioral specifications of input-output signals (...) expressed in a linear temporal logic (...) with non-linear arithmetic constraints and timing constraints on signals", compiling it into "a large number of industrial field-level devices". The tools synthesis algorithm uses a "pseudo-Boolean" abstraction to represent data constraints and translates timing constraints into linear temporal logic. G4LTL-ST assists engineers in formulating realizable control problems by proposing appropriate restrictions on the environment's behavior based on failed synthesis attempts. This tool does not directly incorporate GRAFCET.

- **HyPLC** - Created by Luis Garcia, Stefan Mitsch and Andre Platzer [17], HyPLC is a tool that links formal verification with industrial control programming. It works with hybrid programs that combine continuous physical dynamics and discrete control logic, and translates them into PLC code used in real systems. The process starts by modeling and formally verifying a system as a hybrid program using theorem proving based on differential dynamic logic in KeYmaera X. After verifying safety and correctness properties at the model level, HyPLC generates executable PLC code from the verified control logic. It can also translate existing PLC code back into a form suitable for verification when provided with the system's physical dynamics. In this way, HyPLC preserves the properties from the verified model through to the implemented controller, bridging formal verification and practical PLC deployment in safety-critical systems.

- **LLM4PLC** - Created by Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros Araya, Oluwatosin Ogundare and Mohammad Abdullah Al Faruque [19], LLM4PLC is a framework that improves the use of Large Language Models (LLMs) for generating PLC programs used in industrial automation. Instead of relying solely on an LLM to produce code, LLM4PLC uses an iterative generation and verification process. The LLM first generates PLC code, which is then checked by external tools such as grammar checkers, compilers, and model checkers (SMV verifiers). If errors are found, the feedback is provided to the LLM, allowing it to refine the code until it satisfies the required syntax and correctness checks. The framework also enhances code generation through prompt engineering and fine-tuning techniques, including the use of Low-Rank Adaptation (LoRA) models. Experimental results from their study show that the framework increases the success rate of valid code generation and produces programs that are substantially better suited for industrial control applications than those generated by standalone LLMs.

- **LLM4SFC** - Created by Ofek Glick, Vladimir Tchuiev, Marah Ghoummaid, Michal Moshkovitz and Dotan Di-Castro [25], LLM4SFC is a framework that uses LLMs to automatically generate SFCs from natural-language descriptions of industrial workflows. To address the challenges of generating graphical PLC programs, LLM4SFC represents SFCs in a simplified structured textual format, which preserves the chart topology and embedded ST actions. It further improves generation quality through fine-tuning, Retrieval-Augmented Generation (RAG), and a structured decoding process that prevents the LLM from producing invalid syntax, ensuring compatibility with industrial toolchains. By combining these techniques, LLM4SFC generates syntactically valid and executable SFC programs from natural-language specifications. Experimental results on real-world manufacturing datasets demonstrate high generation success rates, showing its potential to automate the development of graphical PLC programs for industrial automation.

Unfortunately, adapting grafcet2plc to the required use case proves itself to be more complex than necessary as it is developed "for an old version of CADEPA and a Siemens Simatic S7-200" which is undesirable.

G4LTL-ST was not chosen as there is seemingly no way of accessing it.

Given that HyPLC does not natively support GRAFCET usage, requiring some work-around, and is seemingly inaccessible, it was not chosen for use in this work.

LLM4PLC is not suitable for this works purpose because it is designed to generate PLC code from natural-language descriptions, rather than translating GRAFCET models.

Its primary objective is to improve the correctness of LLM-generated PLC programs through iterative verification, not to preserve the semantics of a GRAFCET specification during translation into an IEC 61131-3 language. Moreover, it would add a great deal of complexity to an objectively simple task.

LLM4SFC was also not chosen as it does not address the required translation task. Although it targets the IEC 61131-3 SFC language, it generates SFC programs from natural-language workflow descriptions instead of converting existing GRAFCET models, similarly to the previous tool. Since it is a code generation framework rather than a model-to-model translation tool, it would require some interfacing program to describe a given GRAFCET model if even compatible.

In the end, after considering the aforementioned tools, the option to create an IEC 61131-3 translation toolchain was chosen. Most of the previous tools are far more complex than what is required, this work also intends to lower complexity as much as feasible as otherwise the toolchain could possible become far too convoluted. In addition to this, some of the tools are either outdated, inaccessible, or just do not perform the actions required of them, i.e. translating from GRAFCET to IEC 61131-3, preferably to the ST language as described in section 3.4.

3.1.3 Target Tool for IEC 61131-3 Code

Regarding the tool chosen as a recipient of the generated IEC 61131-3 code, alternatives beyond the one suggested in the dissertations topic page were not sought out, meaning CODESYS was the chosen platform. This decision was made given multiple points related to my experience with this particular tool:

1. Experience with creating programs using the IEC 61131-3.
2. Experience with variable exposure using the OPC-UA module and knowledge on how to integrate it with MQTT.
3. Experience in debugging and testing the created programs.

3.1.4 Other Tool Considerations

It is worthy of note that some of the works researched for the SoA make use of certain tools that were considered, yet, ultimately, not implemented. The works [14, 15] use the Eclipse Modeling Framework (EMF) [6], an open-source Java-based software solution, where a GRAFCET editor was created with the intent of generating grafcets that then, after being passed through a Graphical Modeling Framework (GMF) with the appropriately created meta model, leads to the generation

of an XML Metadata Interchange (XMI) file, all within the EMF environment. In [14], this final XMI file, can then be subjected to syntax verification given the existing syntax check file, to ensure GRAFCET correctness. On the other hand, [15] redirects the XMI file into Acceleo [24], a model-to-text (M2T) transformer incorporated into the EMF environment as a plug-in, which then generates the IEC 61131-3 code in ST. Additionally, [22] uses EMF as well, where it is simply stated that it is used for "the syntactical check and the transformation of GRAFCET to ST code".

In the end, the use of EMF and/or Acceleo were discarded. This decision was made on the grounds that having to learn how to program and utilize these new tools, instead of relying on already understood ones, would increase the development time required for the desired toolchain.

Another point of consideration was GRAFCET-Studio's CODESYS libraries, which although being unable to directly translate grafcets created in GRAFCET-Studio into IEC 61131-3 code, they can integrate with CODESYS via an OPC-UA connection - a process described here.

This however was also discarded, as it goes against the desire to keep the project open-source or at least with as much inward visibility as possible, not acting akin to a "black-box", where no information or processes are able to be analyzed from the outside.

3.2 GRAFCET Interpretation Algorithm Rules

The GRAFCET standard (IEC 60848) contains many unfortunate ambiguities. A deterministic interpretation of GRAFCET is essential to ensure that the control system behaves in a predictable and unambiguous manner. For any given system state and set of input conditions, the same sequence of steps and transitions will always be executed, which is a critical requirement in industrial automation. This determinism eliminates ambiguity in transition firing and step activation, thereby reducing the risk of unexpected or unsafe behavior. Furthermore, it facilitates formal verification, systematic testing, and easier maintenance of the control logic, all of which are particularly important in the context of complex, safety-critical, and industrially deployed systems. As such, a set of rules for the GRAFCET interpretation algorithm are required, which there are many of.

In [21], Mross et al. seeks to dispel these ambiguities by proposing rules for a deterministic interpretation:

- Initialization and Evolution: Initial steps are marked as "activated" but actions execute only in the first evolution; no events are processed during initialization.
- Simultaneity and Conflicts: System variables are copied at the start of each evolution. All condition evaluations and state changes within that evolution are based on this copy, preventing read-write conflicts and simulating simultaneous execution.
- Event Handling: An input event, triggered by a variable change, is valid only for the first evolution in a transient run. Internal events, caused within a run, are valid for the subsequent evolution.

- Hierarchical Processing: Partial GRAFCETs are processed in order of hierarchical depth (highest first). Forcing orders and enclosures are applied immediately upon step (de)-activation, ensuring preemptive behavior.
- Macro-steps: They are treated as syntactic sugar, replaceable by their expansion chart without added memory semantics.
- Stable Output: Continuous actions update output variables only when a stable situation is reached.

In other words, at the beginning, the initial steps are marked to be activated and internal variables are processed (inputs read and set), followed by the creation of a copy of the internal and step variables. Then, for every available partial grafcet, checks if transitions are clearable, marks steps with the appropriate state outcome and processes event actions. After this, step states are evaluated, forcing orders, macro step, and enclosing step states are evaluated and the appropriate stored actions are executed. Finally, it checks whether a stable state has been reached, if not it returns back to the copy variables steps and goes from then on, otherwise it goes on to check continuous actions and assigns outputs, returning to the copy step and repeating.

In [5], René David and Hassane Alla formalize the core execution semantics of GRAFCET through a canonical interpretation algorithm designed for operation in the fundamental mode. This algorithm is essential as it defines the behavioral specification that any correct implementation, including an automated transformation into executable code, must replicate. The algorithm meticulously governs the transition between stable situations. Its operation is triggered by external input events and is defined by several critical mechanisms:

- Iterated Firing to Stability: Upon an event, the algorithm enters a loop to simultaneously fire all fireable transitions, iterating through transient situations until a new stable situation is reached. This entire evolution is considered instantaneous.
- Differential Action Handling: It enforces the distinct semantics of impulse actions, executed immediately upon step activation, even during transience, and level actions, updated only once a stable situation is achieved.
- Stability as a Prerequisite for Input Scan: A pivotal contribution is its rigorous justification for why a new input scan must only occur after reaching stability. The provided counter-example demonstrates that evaluating new inputs mid-evolution, as in a naive implementation without a stability search loop, leads to non-deterministic and hazardous behavior, such as the simultaneous activation of contradictory output commands.

Provost et al. addresses the lack of formal semantics in the IEC 60848 GRAFCET specification language, which hinders formal verification in [31]. The authors propose a method to define formal semantics by translating a non-timed GRAFCET model into an equivalent Finite State Machine (FSM) called a Stable Location Automaton (SLA). Key contributions include the formal

definition of a GRAFCET is formally defined as a 4-tuple (inputs, outputs, charts, initial steps) and of an SLA as a 5-tuple (inputs, outputs, stable locations, initial location, evolutions). The semantics encompass core GRAFCET elements: steps, transitions with Boolean conditions, and both continuous and stored actions. A stable location captures a set of active steps and emitted outputs under specific input conditions. The construction algorithm for the SLA uses symbolic Boolean calculus to compute all sets of fireable transitions from a stable location, determine reached situations, and calculate emitted outputs, thereby avoiding state explosion. The approach is limited to non-timed GRAFCET models.

Bayó-Puxan et al., in [2], explore a methodology for a GRAFCET compiler, focused on microcontrollers, that turns grafcets into C code. For this, the authors elaborate a set of compilation rules where a PLC variable scan analogue, Gscan, is to be used. It begins with the system variable initialization, then Gscan is used to capture and retain the system input values. This is then followed by the calculation of the new internal state, which entails the managment of system timers, the receptivity evaluation, a copy of the system state is made and only then is the grafcet evolution performed. After which, the stored actions are executed and, crucially, the system's state is evaluated to check if there were changes, if yes it returns to the system state copy step, otherwise continuous actions are triggered and the systems outputs are enabled. With the outputs set, Gscan then begins a cycle anew, reading the input variable values and so on.

Both of the previous works [5, 21], proved themselves to be pivotal in the development of the toolchain as they establish the base interpretation algorithm, similar to that used in [33]. On the other hand, the study developed in [31] will not be considered, as this works' main concern does not involve verification of model correctness. Moreover, it would require a transposition of GRAFCET models into SLAs, which would add to the toolchain's complexity and increase the difficulty in its development, both of which are undesirable. The handling of nested hierarchical structures seems to go unaddressed and, as previously stated, non-timed GRAFCET models are excluded from the study, this is problematic as many programs may depend on such structures. Finally, the work in [2], although useful to understand the overall structure of a possible GRAFCET interpretation, as it is presented plainly, lacks mention of how to handle many of the required structures and concepts, such as grafcet hierarchies, nesting and hierarchy related components (enclosing and macro step expansions).

3.3 Meta Model for GRAFCET

As outlined in chapter 1, one of the objectives of this dissertation was to, additionally, provide a meta model. It was concluded that, after a couple of iterations, the model should present it self in as simple a way as feasible possible, so that it may not clutter the toolchain development.

In [32] Provost et al. provides a formal, automated method for exhaustive conformance testing of logic controllers specified with GRAFCET. Its core contribution is a semantics-preserving, two-phase transformation of a GRAFCET model into an equivalent Mealy machine. First, a Stable Location Automaton (SLA) is built, formally capturing all stable states (active steps and outputs)

and the Boolean input conditions causing evolution. Second, the SLA is translated into a deterministic Mealy machine by encoding Boolean conditions as minterm-based input events. While this work focuses on synthesis (GRAFCET to IEC 61131-3 code), Provost et al. solve the complementary verification problem.

On the other hand, Mross et al. in [22] define a semantics-preserving transformation from GRAFCET to the Guarded Action Language (GAL), a transition system format suitable for model checking. This translation explicitly supports hierarchical structures (macro steps, enclosing steps, forcing orders) that are often omitted in translations to formalism like Petri nets. An "evolution manager" within the generated GAL model orchestrates the GRAFCET execution semantics, including the handling of transient situations and simultaneous actions. A full GRAFCET meta model is not shown; however, it seems to take mostly after the model proposed in [14] (described ahead), with some additions made regarding the GAL implementation.

In [28], Pardo Martinez and Ferreiro Garcia design and implement a virtual machine with the intent of playing GRAFCET models, for which "An object-oriented static meta-model for the Grafcet language is also introduced". While useful as a basis for the meta model in this work, the model of [28] contains multiple aspects that ultimately led to it not being chosen, aspects such as the lack of most action related elements (with the exception of continuous actions) and the lack of synchronization and parallelization elements.

A more promising implementation of a GRAFCET meta model, more in line with what is demanded of this work, would be that of Julius et al. in [14]. This works intends to create a meta model for GRAFCET which is both accurate, abiding by high quality standard which prevents errors in modeling, and complete, implementing every single aspect of the norm. This meta model is then later applied to the EMF for model syntax checking purposes for grafcets, these being created in an in-house editor within the same framework. The work presents an illustration of the model, figure 3.1, where the multiple elements of GRAFCET and their relations are present in their entirety. It critically displays the relationships with more troubling aspects of the norm, sometimes discarded or omitted by other authors, such as the partial grafcets, both types of expansions and the types of conditions.

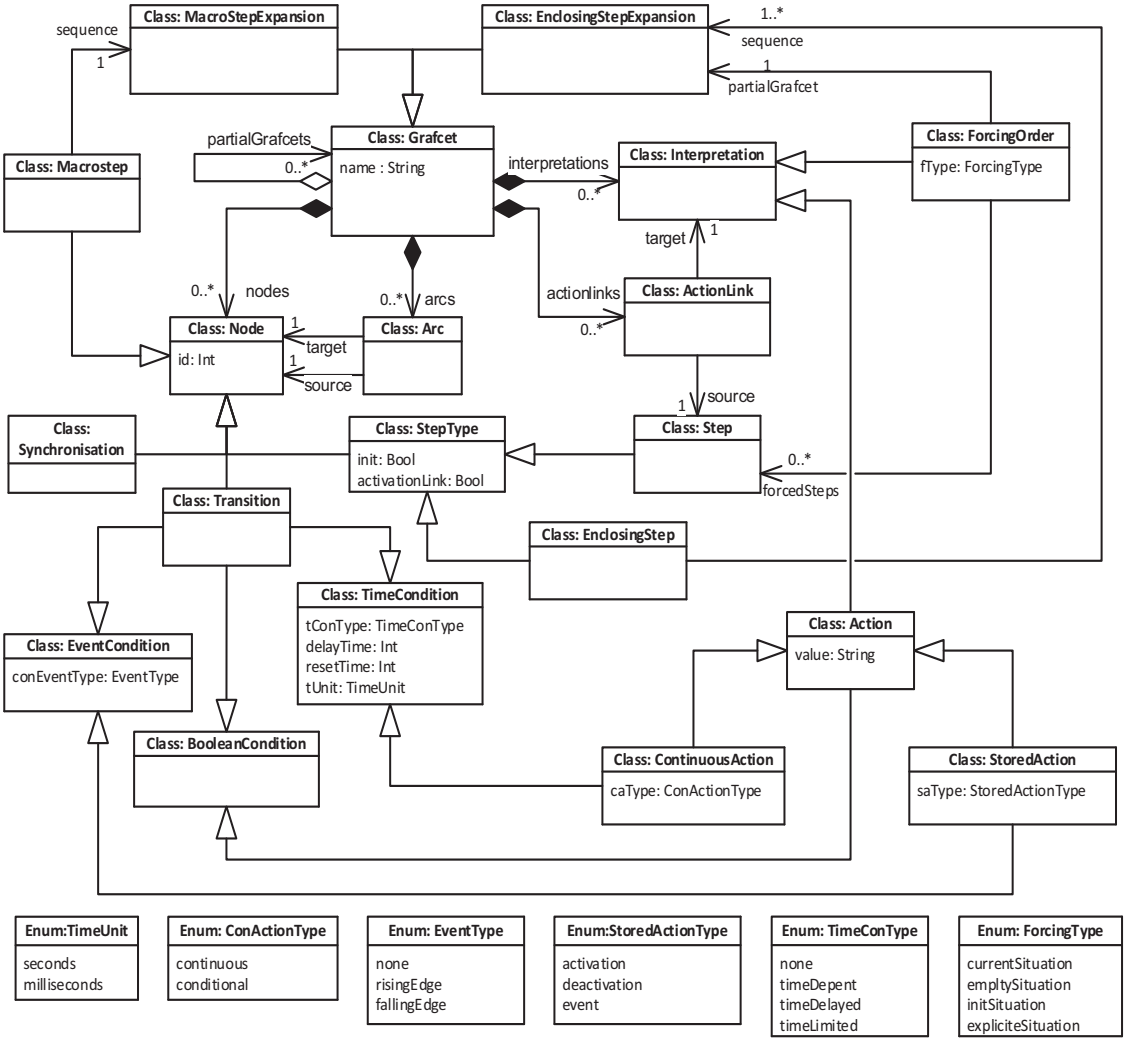


Figure 3.1: Meta model as presented by [14].

The implementation of the models in [32] will not be used, even considering that the mealy machine implementation is an interesting avenue for future works, as for the purposes of this work, this level of verification is not required, as it is assumed the user has a correct GRAFCET, and it would require extensive work to incorporate mealy machines into a system that ultimately demands IEC 61131-3 code. As for [22], the implementation of a GAL system for the objectives of this work would be both unnecessary and bothersome, especially since model checking was not intended for this work. Additionally, although it is an interesting use case in GRAFCET modeling, the given model takes mainly after [14], addressed below. In addition to this, the model presented in [28] is relatively more complex than what is required for this dissertation, as there are many extra classes, enumerations, variables and packages which are intended to facilitate the development of their virtual machine with the internal program "SFC++"; irrelevant for this work. Lastly, the work in [14] was the main work for which the meta model created was based off of. Its rigorous implementation of all components including hierarchical constructs is necessary for the proper development of this work.

3.4 Target IEC 61131-3 Language

Choosing a target language for the PLC transformation is also a crucial point. In [33], Schürenberg evaluates candidate languages against four quality metrics: behavioral similarity, portability, readability, and maintainability. His key conclusion is the rejection of Sequential Function Charts (SFC), despite its graphical and syntactic resemblance to GRAFCET, in favor of Structured Text (ST). He claims SFC suffers from formal semantic deficiencies, particularly in modeling GRAFCETs transient evolution within a single scan, leading to vendor-dependent implementations and ambiguous behavior for hierarchical elements like forcing orders; its modeling of hierarchy, while possible, relies on complex implicit links that obscure the specification's structure. In contrast, Structured Text provides the flexibility to explicitly implement a GRAFCET interpretation algorithm, enabling precise control over execution order, correct simulation of simultaneous and transient behavior, and the direct embedding of hierarchical semantics within portable, modular Function Blocks (FBs), thereby preserving clarity and ensuring cross-platform consistency.

Moreover, in [21], ST is also chosen as its expressiveness in representing complex logical and arithmetic terms from GRAFCET within an automated generation workflow. Its textual nature aligns seamlessly with model-driven transformation, allowing the hierarchical structure of GRAFCET charts to be mapped naturally onto stateful FBs, which retain step activity and communicate via global variables for simplicity. The output is structured in the vendor-neutral PLCOpen XML format, ensuring compatibility with tools like CODESYS and directly supporting the dissertation's goal of automated, portable code generation.

Julius et al. in [16], addresses the loss of hierarchical structure when automatically transforming GRAFCET specifications into IEC 61131-3 PLC code. Prior methods required flattening the model for translation into SFCs, reducing readability and maintainability. The proposed method preserves hierarchy by transforming GRAFCET elements into ST code within FBs. A formal GRAFCET meta-model and a set of explicit transformation rules ensure a one-to-one mapping from specification constructs (steps, transitions, macro-steps, forcing orders) to ST code fragments. A coordinating main program emulates the event-driven GRAFCET semantics on a cyclic PLC. The authors implemented a prototypical toolchain that automates this transformation, demonstrating that hierarchical GRAFCET models can be converted into structured, executable PLC code while maintaining the original organizational clarity.

In addition to this, Julius et al. in [15] presents a model-driven software engineering (MDSE) approach for the automatic transformation of hierarchical GRAFCET specifications into IEC 61131-3 ST code. It extends the authors' prior work [16], by implementing the transformation within an integrated, open-source toolchain and by incorporating time and event constraints. The core of the approach is a static GRAFCET meta-model implemented in the EMF. A graphical editor is generated via the Graphical Modeling Framework (GMF), and the code generator is implemented using the Acceleo model-to-text transformer. This environment takes a GRAFCET model and produces vendor-independent ST code, exportable as PLCopenXML or Siemens SCL.

A key contribution is a refined GRAFCET interpretation algorithm, formalized to adhere to the five standard evolution rules. The algorithm is designed to be executed within a single FB for the entire specification, a variant chosen to avoid the need for global variables when managing hierarchical elements like forcing orders. This algorithm iteratively evaluates enabled and fireable transitions, manages transient evolution to find stable situations, and executes stored and continuous actions to ensure semantic equivalence. The article demonstrates that the model-driven toolchain successfully preserves hierarchical structures during transformation, providing a practical and extensible foundation for automatic code generation that aligns with the principles of model-driven engineering in automation.

Considering that a majority of the works studied for this SoA research make compelling and persuasive points regarding their use of ST over other IEC 61131-3 languages, ST was selected for this work as well. Moreover, [33] proposes detailed and well structured transformation rules used in the process converting GRAFCET into ST code, which will serve as a helpful base for my own, thus making the choice of ST more appealing. Even when not considering the SoA research, ST proves itself to be the most similar to programming GRAFCET logic using a non-61131-3 language, such as C++ for example, which is a field of study that is included in my curriculum (mostly simple grafcets were tackled compared to this work).

3.5 Framing and Contribution to the SoA

The literature reviewed throughout this chapter shows that significant progress has been made in the formalization, interpretation and automatic transformation of GRAFCET models. Existing works provide deterministic execution semantics for IEC 60848 ([2, 5, 21, 31]), define comprehensive meta-models capable of representing hierarchical GRAFCET constructs ([14, 22, 28, 32]), and propose several approaches for automatically generating IEC 61131-3 ST code ([15, 16, 21, 33]). Together, these works establish a solid theoretical and practical foundation for model-driven development of industrial control applications.

Nevertheless, the reviewed approaches also reveal several limitations that further motivated the development of this dissertation. Some of the available solutions are embedded within large model-driven engineering environments, particularly those based on the EMF. While these environments provide powerful capabilities such as syntax verification, model checking and automated code generation, they also introduce considerable implementation complexity and require specialized tooling. Furthermore, some of the proposed toolchains are either unavailable or outdated and do not directly support GRAFCET-Studio usage unless an interface between them is created.

Although previous authors propose complete GRAFCET meta models, they lack certain semantic associations that lead to the models being more closely related to the IEC 60848 norm, for instance, the connected grafcet concept. Additionally, the connections made for certain models are specialized toward use with certain tools, making them more specific than desired, not being able to be adopted by other tools without changes being made. Some of these connections could

also benefit from being made more explicit, so the flow from component to component is more apparent.

Likewise, existing transformation methodologies show that ST is the most suitable target language for preserving GRAFCET semantics. The reviewed literature consistently argues that ST provides sufficient flexibility to explicitly implement deterministic interpretation algorithms while maintaining portability across IEC 61131-3 platforms. This observation strongly influenced the architectural decisions adopted in this dissertation.

Based on these observations, this dissertation positions itself as a practical model transformation approach that can be considered an extension of what was done by Schürenberg in [33]. Rather than proposing new execution semantics or formal verification methods, this work builds upon the deterministic interpretation algorithms established in the literature and adapts them into a standalone software toolchain intended to minimize external dependencies while maintaining semantic fidelity to IEC 60848.

The main contributions of this dissertation, in no particular order, can therefore be summarized as follows:

- It addresses the lack of GRAFCET-Studio specific translation toolchains, whilst still being able to interface with other programs, so long as they are implemented via a simple code change.
- The meta model used is designed in such a way that it includes all relevant IEC 60848 constructs including those of a more semantic nature and attempts to make the overall model clearer and easier to understand while remaining independent from other frameworks.
- The communication protocol OPC-UA is used to fill the gap when it comes to the lack of IoT capabilities and overall visibility of variables.
- Provides extensive documentation regarding the relatively exhaustive case study related testing performed on the toolchains results - the source GRAFCET-Studio grafcet is tested comparatively to the created code running on the CODESYS virtual PLC.
- Provides not only slightly more refined GRAFCET component translation rules to IEC 61131-3 constructs, as well as small additions to the way the GRAFCET rules were applied at the destination program.

Compared with previous works, this dissertation intentionally prioritizes simplicity and ease of implementation over comprehensive verification capabilities. Instead of relying on complex model-driven engineering environments, it demonstrates that a lightweight standalone architecture is sufficient to automate the transformation from GRAFCET specifications to executable IEC 61131-3 ST while preserving the execution semantics required by industrial control applications.

Chapter 4

Toolchain Architecture

This chapter begins by presenting the architecture proposed in order to solve the given problem. Following this comes the section which addresses the GRAFCET interpretation algorithm. Afterwards, the meta model is shown, starting by outlining its objectives and finishing by giving its description. Additionally, the overall structure for the final PLC program is presented. The chapter finishes with a summary.

4.1 Proposed Architecture

The proposed architecture for this project's toolchain consists of two sequential tools organized as a processing pipeline. The first tool transforms the source GRAFCET model into intermediate artifacts, which are then consumed by the second tool to generate the final outputs. Additionally, a third tool, a simple script, not only calls for the sequential execution of both of the previous tools, but also handles the file importing into the CODESYS environment.

Figure 4.1 represents the proposed toolchain architecture.

The first stage imports the source grafcet model (A), performs pre-processing with its internal GRAFCET model in order to create the missing component-component connections. From this process, it generates two intermediate artifacts, one for the GRAFCET components (B) and the other for its symbols (C), which serve as inputs for the subsequent stage.

The second stage imports the intermediate artifacts (B and C) and constructs a meta-model-based GRAFCET representation. It translates the semantic information contained in B and formats both B and C to generate the final output artifacts, the IEC 61131-3 project file (D), and the symbol configuration file (E).

The final stage of the toolchain consists of an automation script responsible for integrating the generated artifacts (D and E) into the CODESYS environment.

Moreover, this last stage can instead be substituted by the use of another script that, instead of importing the projects into CODESYS, simply executes the two main tools in sequence and leaves their outputs in a specific directory, since file D should be able to be utilized by any program that accepts PLCOpenXML-structured files.

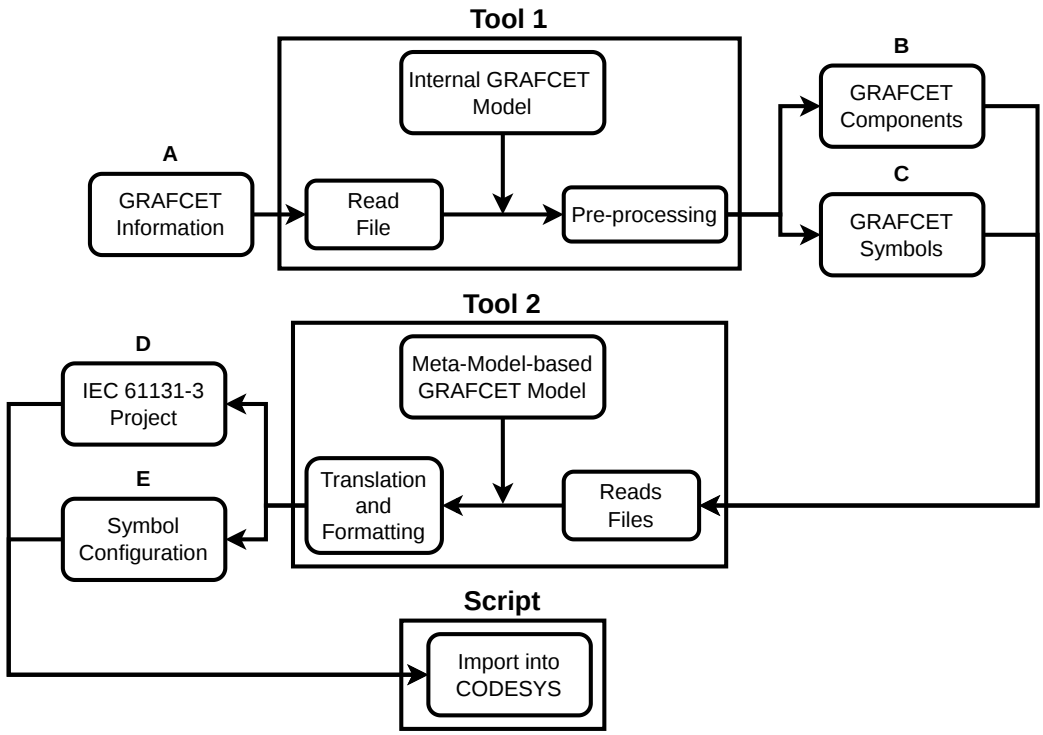


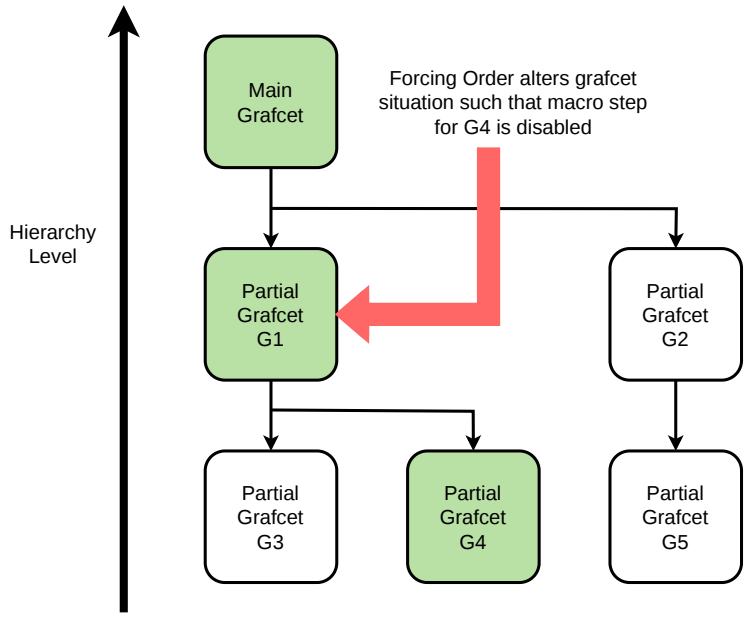
Figure 4.1: Proposed architecture for toolchain.

4.2 GRAFCET Interpretation Algorithm

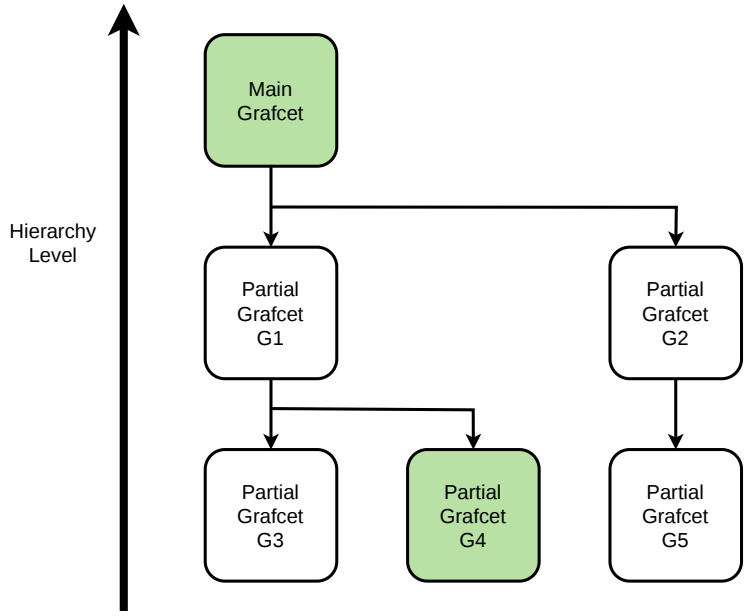
These next paragraphs address the GRAFCET interpretation algorithm developed with the aid of the aforementioned related works in chapter 3 - described in figure 4.3.

This algorithm, also heavily based on what is described in [33], provides the logical basis with which the generated IEC 61131-3 codes will be executed during runtime on their destination program. Moreover, the entire algorithm respects the grafcet hierarchies, by executing the highest order grafcet processes first and going down until reaching the lowest grafcet in the hierarchy.

This grafcet ordering process proves itself to be important as hierarchy errors may otherwise become apparent in the IEC 61131-3 ST translation of the grafcet. For example, if a forcing order changes the situation of a particular partial grafcet, which could lead to a macro step changing its state to inactive, all lower hierarchy grafcets that belong to that macro step expansion "branch" are to be deactivated, if the proper partial grafcet hierarchy is not achieved then this previous situation may not occur, concluding in an improperly built final translation - see figure 4.2 below, where green filled squares represent active grafcets and white filled squares inactive grafcets.



(a) Normal situation is affected by forcing order.



(b) Imperfect result, illustrating poor algorithm.

Figure 4.2: Grafcet hierarchy problem example.

The GRAFCET interpretation algorithm, illustrated in figure 4.3, starts as most other related algorithms, with the initialization of the relevant variables, be it input or output or simply internal state variables, and setting their values, including the initial-situation-participating step state

values.

Following this, the input values are read and the rising/falling trigger variables and the timed conditioned variables are evaluated, with these latter evaluation seemingly not being considered by other papers on the topic. These trigger and timed variables, but especially the triggers, prove important for the full implementation of the GRAFCET toolkit, be it for stored actions or event management.

For the first runtime cycle only, an additional check is provided due to the initial situation steps, where the stored action activations are checked, otherwise they will remain false until subsequent initial step activation are provided, an undesirable outcome where the rules of GRAFCET are being broken.

The algorithm now checks whether the clear condition for each transition are true, if this proves correct then they are marked for activation, not yet being activated; otherwise the program moves onward.

Now the program enters a loop (ignoring the main task loop that had to begin to allow for runtime execution), where a second transition evaluation is performed, now checking if they are marked for clearing and if a step before it is active; if true the transitions are fired. After the firing of the transitions, the states of the steps (before and after) are accordingly updated and the stored actions associated to these updates (and to any possible event for the event driven actions) are executed if conditions are true. This marks the end of this loop, where the algorithm evaluated whether an additional loop cycle would result in a change in the GRAFCET situation; if this is proven to be true, the program will once more loop this section, beginning with the second transition check onward. This loop check helps combat the transient evolution ambiguities, providing a space where these can be resolved, reaching a "final" state, before the rest of the program may evolve, not changing transition clear conditions nor continuous action states

If on the other hand this final loop check is proven false, the algorithm continues to the final step, the updating of the continuous action states, after which another program cycle begins, reading the input variables and so on and so forth.

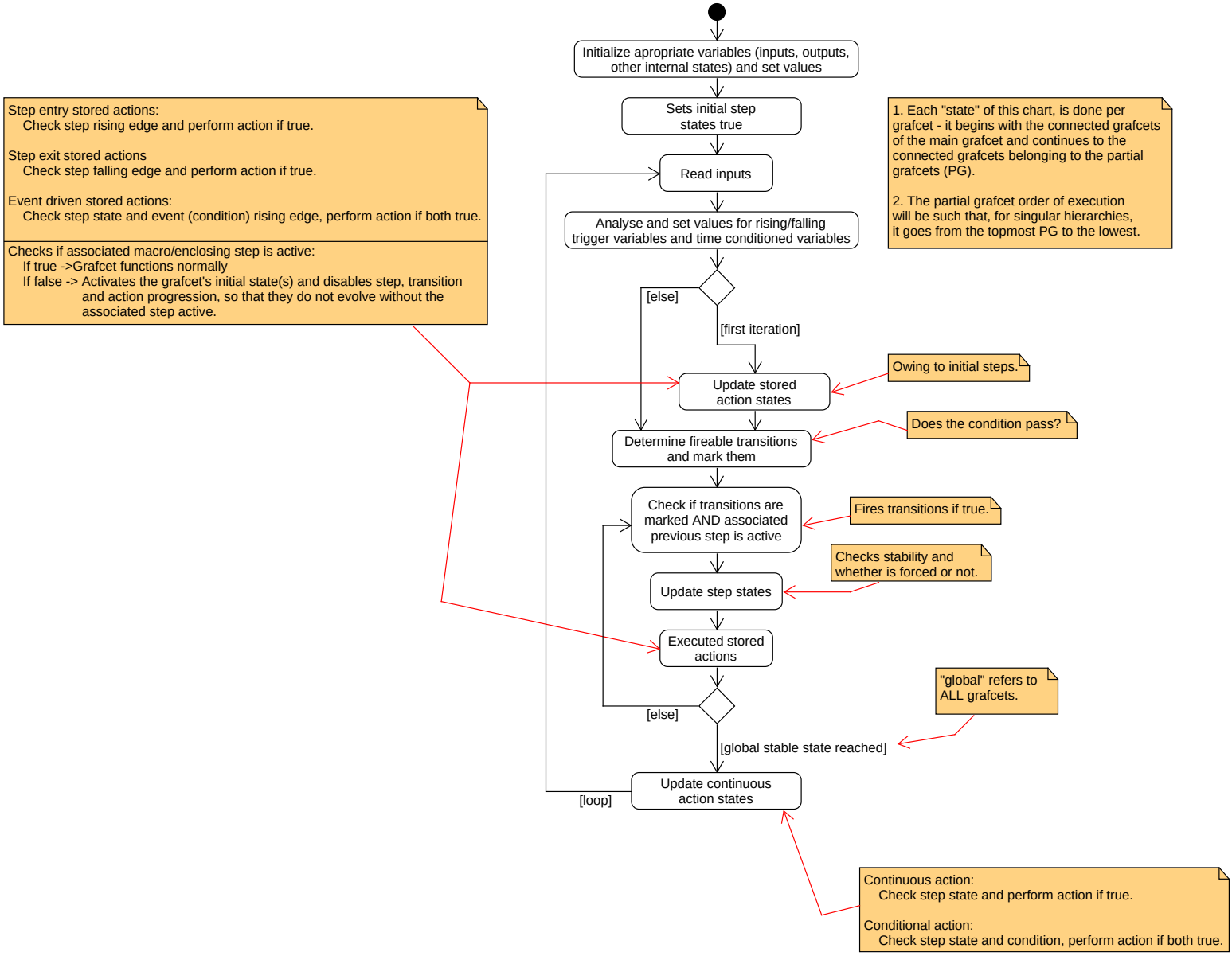


Figure 4.3: GRAFCET interpretation algorithm.

4.3 Meta Model

So as to guarantee that the aforementioned interpretation algorithm is properly implemented into the translation algorithms of the toolchain, a meta model representative of GRAFCET's elements and its constructs was required.

This section first outlines the objectives underlying the development of the proposed meta model.

It subsequently presents a detailed description of the meta model designed and implemented as part of this dissertation, together with a diagram illustrating its complete structure and the relationships between its constituent elements.

4.3.1 Objective

The meta model presented in the following section, subsection 4.3.2, was created with clear objectives.

Apart from its main purpose of helping support the GRAFCET interpretation algorithm. It strives to ensure that the internal data models used by the toolchain, stored in its allocated memory, faithfully represent the according elements from the IEC 60848 standard, acting as a "data check", where any given element is compared against the structure presented in the meta model, with data inconsistencies being highlighted, for instance, checking whether a given element's type correctly matches.

The model additionally served as a guide when building the algorithms that handle grafcet elements, functioning as a straightforward "look-up table" to guarantee that the algorithms do not have any per-element interaction errors. Moreover, having such a model will facilitate the integration of this toolchain with possible future tools, as it allows for other GRAFCET models to be easily accommodated by simply transposing the existing elements from one to the other, the rest of the code after the meta model checking will handle the translation accordingly with no additional code required.

4.3.2 Description

A meta model is an abstract representation that defines the concepts, elements, relationships, and constraints of a particular modeling domain. Rather than describing a certain system, it specifies the structure and rules that individual models must follow.

As mentioned in chapter 3, this works' meta model, demonstrated in figure 4.4 below, shares many similarities with that of [14].

The model contains two auxiliary classes, "**Base**" and "**Tag**", the latter being derived from the former. The class "Base" contains an integer variable "id", which represents an identification number, unique across all objects. The class "Tag" contains a string variable "name", which represents the name of a certain object derived from it. Furthermore, the classes "Grafcet", "PartialGrafcet" and "Node" all derive from "Tag", thus having the variables "id" and "name", whilst the classes "ConnectedGrafcet" and "ActionType" derive only from "Base", having only "id".

The main part of the model begins with the class "**Grafcet**", this represents the grafcet with the highest order in the hierarchy. The class must contain at least one "PartialGrafcet" or one "ConnectedGrafcet" and can contain an infinite number of combinations of both.

The "**Partial Grafcet**" class, describes a sub-division of the main "Grafcet" chart, containing at least one "ConnectedGrafcet" associated; whilst the "**ConnectedGrafcet**" class denotes the element-by-element association - characterized as "Node" here - where there is "always a continuity of links" [12], with at least one "Node" class present.

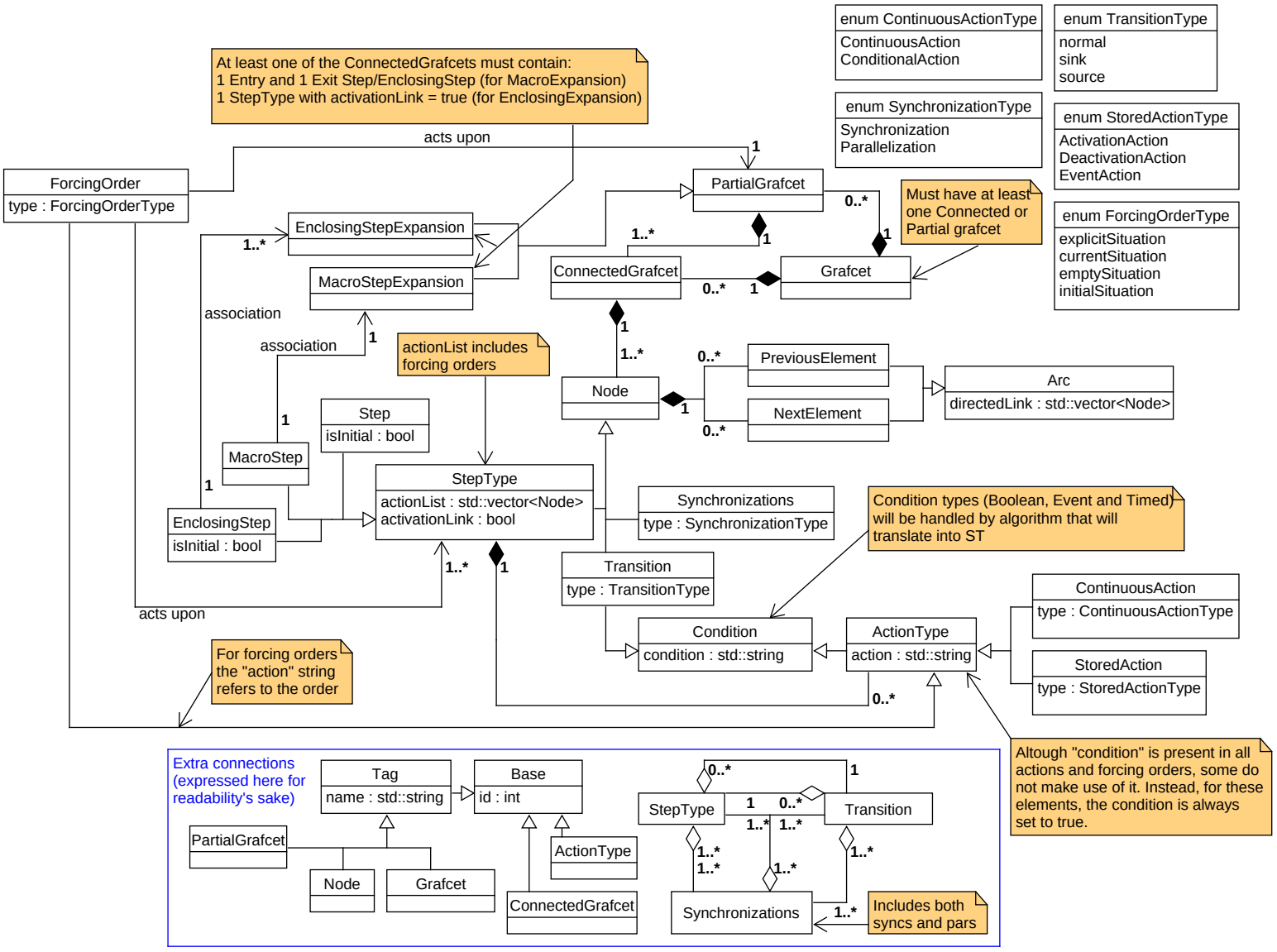


Figure 4.4: GRAFCET meta model.

The class "**Node**" represents the elements themselves. Furthermore, the connections between theses elements are ensured by the "**NextElement**" and "**PreviousElement**" classes, both are derived from the class "**Arc**", which contains a vector of "Node" entities - "**directedLink**". The derivative classes of "Node" are as such:

- "**StepType**" class - Represents all step types. "**actionList**" is a vector of "Node" entities containing all associated actions (including forcing orders). "**activationLink**" is a boolean representing whether a step has or not an activation link, if belonging to a enclosed grafcet. Additionally, objects of this class may have a "directedLink" from one object to one or multiple "Transition"s and one or multiple objects to one or multiple "Synchronizations". "StepType" subdivides into:

  - "**Step**" class - Represents simple steps. "**isInitial**" denotes whether the step is an initial step or not.
  - "**MacroStep**" class - Represents macro steps.
  - "**EnclosingStep**" class - Represents enclosing steps. "**isInitial**" denotes whether the step is an initial step or not.

- "**Synchronizations**" class - Represents both synchronizations and parallelizations, whether the object is one or the other is highlighted by the enumeration variable "**type**". In addition, objects of class "Synchronizations" may be connected from one or multiple objects to one or multiple "StepType"s and to one or multiple "Transition"s.

- "**Transition**" class - Represents all types of transitions. The "**type**" enumeration variable represents the transition type, which take the value of "**normal**", "**sink**" if there is no following element or "**source**" if there is no previous element. Moreover, objects from this class may connect from one object to none or multiple "StepType"s and from one or multiple objects to one or multiple "Synchronizations".

The "**Condition**" class contains the string variable "**condition**", which, as the name implies, represents a certain condition that must become true for a certain check to pass. The type of condition (Boolean, Event and Timed) are to be determined by the toolchain at the time of translation according to the contents of the the variable and the associated class. Both the aforementioned "Transition" and the "ActionType" classes derive from this "Condition" class.

The "**ActionType**" class encompasses all types of actions, containing an "**action**" string variable. This variable represents the action content to be performed by the associated action element. Any given "ActionType" belongs to one and only one "StepType". The actions associated with this class are:

- "**ContinuousAction**" class - Represents all continuous actions. The "**type**" enum variable can be "**ContinuousAction**" or "**ConditionalAction**", depending if the action requires only the associated step to be active or both the step active and an extra condition.

- "**StoredAction**" class - Represents all stored actions. The "**type**" enum variable can be "**ActivationAction**", "**DeactivationAction**" or "**EventAction**" for stored actions that activate on step entry, step exit or given an external event whilst the step is active.

- "**ForcingOrder**" class - Represents all forcing orders. For this class, the variable "**action**" refers to the order and not an action. The "**type**" enum variable can be "**explicitSituation**" for activating specific steps, "**currentSituation**" for "freezing" the current grafcet situation, "**emptySituation**" for deactivating all grafcet steps and "**initialSituation**" to activate the initial situation only. Additionally, this class acts upon a singular "PartialGrafcet", affecting at least one "StepType" from said partial grafcet.

The "**EnclosingStepExpansion**" class represents the expansion of enclosing steps. This class is derived from "PartialGrafcet" and is associated to a singular "EnclosingStep". It must have at least one step of type "Step", "MacroStep" or "EnclosingStep" with the variable "activationLink" valued true.

Similarly, the "**MacroStepExpansion**" class represents the expansion of macro steps. This class is derived from "PartialGrafcet" and is associated to a singular "MacroStep". It must have two steps (entry and exit steps) both of either type "Step" or "EnclosingStep".

4.4 PLC Program Structure

In addition to the interpretation algorithm, the overall program structure is also derived from this same work, [33], where a multitude of CODESYS constructs are used for this end.

This structure begins with the "PLC_PRG" POU, a simple program type POU using the ST language, where the GRAFCET interpretation algorithm which has just been presented is applied - additional information regarding the application of the algorithm in "PLC_PRG" can be found in subsection 5.3.4.

Any CODESYS program requires a task configuration, which will determine the runtime of the selected programs and their behavior. This previous POU, "PLC_PRG", is, for the purposes of this dissertation, the only one to be associated to the "GrafcetTask", a particular task configuration which controls the runtime behavior of the POU.

Another important construct used are the function block (FB) type POUs, dedicated to representing the main grafcet structure and all the partial grafcets, with one FB being exclusively associated to one grafcet. These POUs make use of methods, which present themselves as functions associated to a certain FB, where an input is given and an output is expected, even if they are null. These methods were mostly created in accordance with the ones exposed in [33]; however, minor changes were made to some of the already defined methods to better represent the interpretation logic and to better preserve the GRAFCET correctness in the IEC 61131-3 translation. Additionally, there are a few extra methods developed to acomodate the additions made in the interpretation logic:

- **triggerHandler** - Handles the generic triggers, reading the input values for the associated trigger variables and determining whether the trigger is be valued at true or false, using for this the existing CODESYS variable types R_TRIG and F_TRIG accordingly.
- **stepTriggerHandler** - Similar to the triggerHandler method in function, being however exclusive to steps, mostly relegated to help the stored action state management method - storedActionHandler - as it depends on the step entry/exit information.
- **timedConditionHandler** - Manages time related constructs, where it checks whether a certain time has elapsed for all the variables contained within, populating the according FB variables with the boolean value of this previous result, making use of the CODESYS variable types TON, TOF and a combination of both.

4.5 Summary

This chapter presented the architecture and conceptual foundations for the proposed GRAFCET-to-IEC 61131-3 translation toolchain. It began by introducing the overall processing pipeline, composed of two sequential transformation stages and an auxiliary automation script, illustrating how GRAFCET models are progressively transformed into the final two output files suitable for integration within the CODESYS environment.

It then goes onto describing the proposed GRAFCET interpretation algorithm, which establishes the execution semantics that the generated PLC program must preserve. Particular emphasis was placed on the correct handling of GRAFCET hierarchies and transient evolution, ensuring compliance with the execution rules defined by the IEC 60848 standard while resolving ambiguities that may arise during runtime. Additionally, the iterative evaluation strategy shown guarantees that each execution cycle reaches a stable GRAFCET situation before continuous actions are updated, thereby preserving the deterministic behavior expected from industrial control applications.

Subsequently, the proposed meta model was introduced as the central abstraction used throughout the translation process. Beyond providing a faithful representation of the constructs defined by the IEC 60848 standard, the meta model serves as a validation mechanism for the internal data structures employed by the toolchain and as a common representation that simplifies the implementation of translation algorithms. Its hierarchical organization and explicit modeling of GRAFCET elements, relationships, and constraints also facilitate future integration with other modeling tools by providing a well-defined intermediate representation independent of the source modeling environment.

Finally, the overall structure adopted for the generated PLC program is presented, detailing the organization of the principal POU, responsible for implementing the execution semantics established by the interpretation algorithm, the task configuration and the function blocks. The additional methods introduced to support trigger management, step activation detection, and timed conditions further demonstrate how the generated software encapsulates the required GRAFCET behavior while remaining consistent with CODESYS programming practices.

Overall, the concepts presented in this chapter establish the theoretical and architectural basis for the implementation of the proposed toolchain, described in the following chapter.

Chapter 5

Toolchain Implementation

This chapter presents information regarding the toolchain developed.

It will begin by going over its layout where the connections between the toolchain components will be discussed. Additionally, an overall usage flow is provided, where an overview of the flow of normal toolchain operation is shown step-by-step.

Next, the first part of the toolchain which acquires the information needed from GRAFCET-Studio, whilst the following second part handles its formatting, code generation and OPC-UA capabilities. Both of these parts shall provide a description of each tool's objective, overall structure and libraries, as well as their respective outputs.

Following this, the objectives and overall description of the CODESYS import script will be highlighted.

This chapter ends with a summary of what was discussed.

5.1 Layout

This section describes the structure of the toolchain's directory, detailing the purpose of each of its components. It also provides an overview of the toolchain's workflow, outlining the sequence of operations and the interaction between its constituent elements.

5.1.1 Directory Structure

The toolchain is organized into multiple directories. These directories and their hierarchical relations are illustrated by the following figure, figure 5.1.

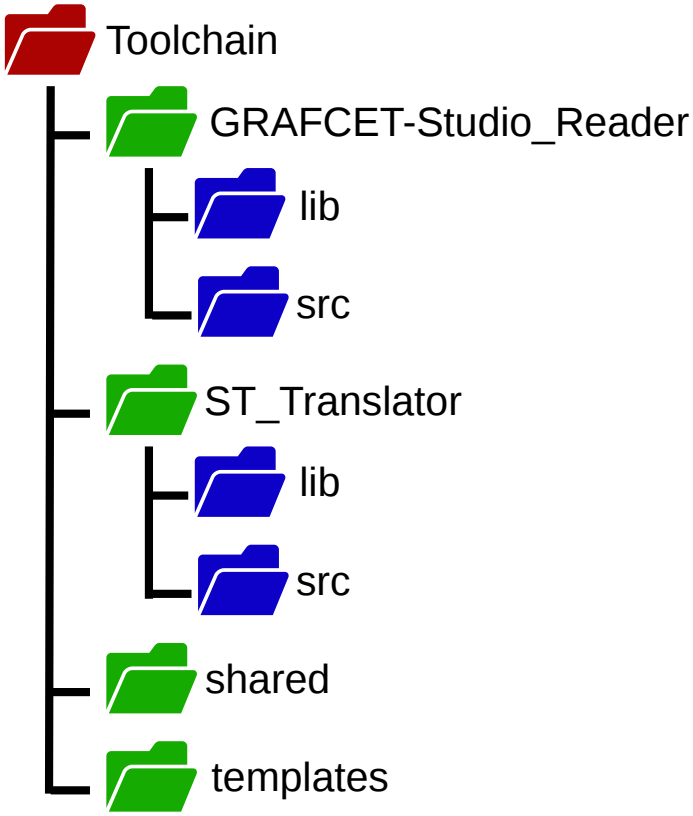


Figure 5.1: Toolchain directory hierarchy.

The main toolchain directory - "Toolchain" - contains the script, written in Python, that imports the finished ST code into CODESYS, see section 5.4, as well as the compile and run batch files used to run the toolchain outside of CODESYS usage, for instance, for simple console debugging. It is additionally subdivided into four directories.

The "GRAFCET-Studio_Reader" directory pertains to the first part of the toolchain, detailed in section 5.2, containing two directories, one for the libraries used - "lib" - and the other for the main code and a "compile and run" batch script for debug purposes - "src".

The directory structure pertaining to the second directory, "ST_Translator", the second part of the toolchain, is much the same to that of "GRAFCET-Studio_Reader". It is detailed in section 5.3. Both of these parts were written in C++.

The third directory - "shared" - will contain all the necessary files consumed and produced by the toolchain programs, including the grafcet save file from GRAFCET-Studio. Whilst the fourth directory - "templates" - holds the template files used in assisting the creation of the formatted PLCOpenXML files.

5.1.2 Usage Flow

The toolchain's usage usage will have particular flow of data and actions, involving the two aforementioned programs and the Python script. These are as such:

1. Use GRAFCET-Studio to create and save a grafcet.
2. Move created grafcet XML save file to "shared" directory.
3. Use CODESYS' python script running functionalities to run the import script.
4. GRAFCET-Studio Reader reads the grafcet XML file.
5. GRAFCET-Studio Reader parses grafcet data and the pertinent variables.
6. GRAFCET-Studio Reader makes the element-by-element associations.
7. GRAFCET-Studio Reader saves two files to "shared" directory - "IntermediateSave.xml" and "SymbolSave.xml".
8. ST Translator reads both "IntermediateSave.xml" and "SymbolSave.xml".
9. ST Translator parses information from both files.
10. ST Translator builds objects according to the metamodel in chapter 4.
11. ST Translator creates the appropriate save files - "CODESYSProject.xml" and "Symbol-Configuration.xml" - according to the templates in the "templates" directory.
12. CODESYS imports the data from these final two files.
13. CODESYS builds the project.

5.2 GRAFCET-Studio Reader

This section begins by outlining the objectives of this component of the toolchain, followed by an overview of the libraries employed to achieve them. It then presents the program's workflow, divided into its main processing phases. Finally, the outputs generated by this component are examined and their purpose and structure are explained.

5.2.1 Objectives

The main objectives of this script are to parse the information read from the GRAFCET-Studio XML save file, make the semantic associations between elements - since, as previously stated, GRAFCET-Studio does not provide these connections - and, finally, save all the element data into a formatted XML file, "IntermediateSave.xml", and the variable data into a different XML file, "SymbolSave.xml", akin to GRAFCET-Studio.

5.2.2 Libraries

There are in total three libraries being used for this particular tool, one borrowed and two made.

The first library, which was borrowed, is called **"pugiXML"**, maintained by Arseny Kapoulkine. pugiXML is a XML processing library for C++ with a DOM (Document Object Model) based interface, allowing for simple and fast reading and writing of XML files.

The second library is **"aux_structs"** which contains the structs, enumerations and functions used to create the appropriate objects associated to the grafcet elements, some components from GRAFCET-Studio itself (namely pages, jumps and connection lines) and symbols. Furthermore, it contains maps that allow for easier data handling from strings to enumerations and vice-versa.

Thirdly, the **"aux_funcs"** library harbors all the functions necessary to make the aforementioned semantic associations. These functions are of the type **"check_X()"** where "X" represents the name of the element being checked for possible connections. It additionally has a function, **"make_association()"**, to perform the associations themselves.

5.2.3 Program Flow

This following description gives more in-depth insight into the flow of the GRAFCET-Studio Reader program presented in subsection 5.1.2. Additionally, figure 5.2 presents this process via a flowchart.

- **Extraction Phase** - Acquisition of the grafcet information.
  - The extraction phase begins by reading the information from the GRAFCET-Studio save file, located in the "shared" directory.
  - It then creates "page" objects, to which the components belong to, via the **create_page()** function.
  - Afterwards the **create_component()** function is utilized to create the individual elements of the grafcets, extracting the relevant information from them, such as: their type, X and Y values, width and height values and more component-specific data.
  - Finally, the **create_symbols()** function extracts the information related to the symbols (i.e. the variables), more specifically, their name, data type and symbol type.
- **Association Phase** - Performing component-component associations.
  - The association phase begins by executing the **check_action()** function, where all the action components, including forcing orders, are associated to their respective step component. This function comes before all other check-type functions as otherwise an error occurs when analyzing the connection lines.
  - Following this, all other components are analyzed with their respective check-type functions. These functions, akin to the check_action() function, utilize the data acquired from the previous phase in order to perform the connections to the appropriate components. These functions are as follows:

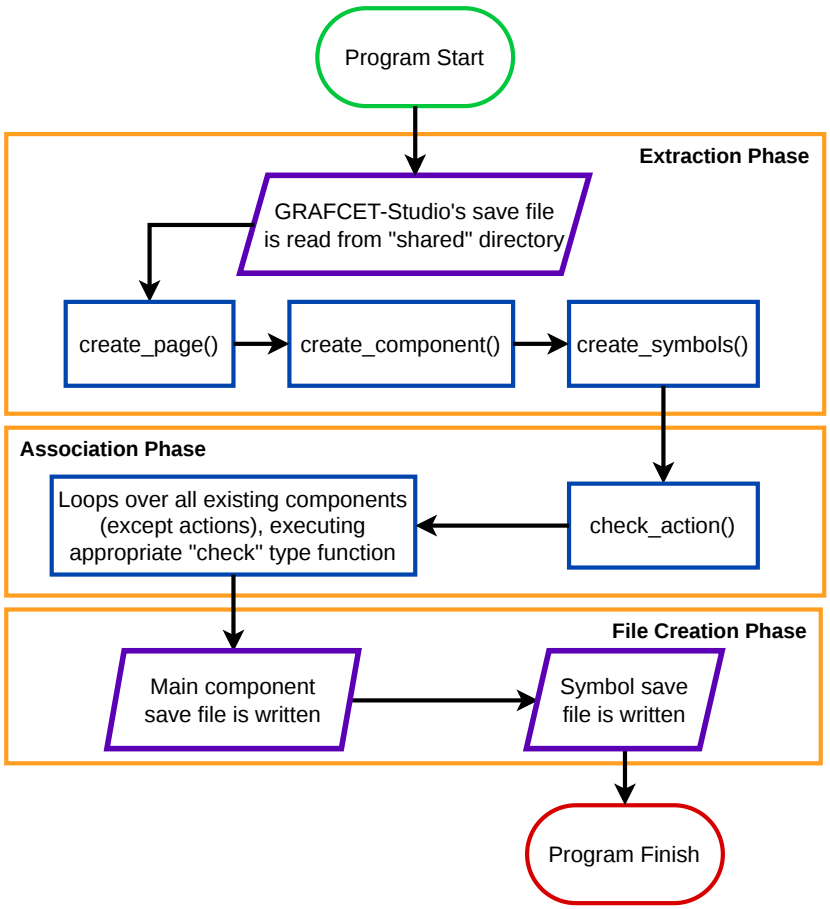


Figure 5.2: Flowchart representing GRAFCET-Studio Reader's program flow.

- **check_connection_line()** - The check_connection_line() function is always called by the other function in this list, where its objective is to find a connection from a source component to a destination component. It does this by iteratively checking the points of connection for the connection lines belonging to a given chain, essentially going from one side of this chain to the other and making the appropriate component associations at the end.
- **check_step()** - This function analyzes four types of component relative to the step: Transitions, Connection Lines, Synchronizations and Parallelizations. All of these components are analyzed to check whether they are previous/next components relative to the step, with the exception of the synchronizations (only previous) and parallelizations (only next).
- **check_jump()** - When it comes to jumps the top side is checked for one of two possible possible connections, either a transition or a connection line(s) connected to a transition, associating the transition with the step to which it points to.

    - **check_parallelization() and check_synchronization()** - These two functions operate in the exact same way, only changing the component type for which they search for on each side (top/bottom). check_parallelization() checks for transitions or connection lines that end in a transition at the top and connection lines which will end up connecting to steps at the bottom. On the other hand, check_synchronization() checks for connection lines that will lead to steps at the top and transitions or connection lines that lead to transitions at the bottom.
    - **check_groupBorder()** - The function's objective is to find the components within the borders of a group border. Then the component being analyzed, if belonging to the border component, have the borders' name, associated border step and the type of border (normal/macro/enclosing) associated to it.

- **Output Creation Phase** - Create output files from component and symbol data, relying on the pugiXML library to handle the XML structuring.
    - In the output creation phase, the output files, created from all of the data acquired and built in previous phases, are written to the "shared" directory. It begins by writing the intermediate save file, responsible for all the information about the pages and their respective components.
    - Following this, the symbol related information file is also saved, containing all of symbols' data.

5.2.4 Outputs

The GRAFCET-Studio Reader tool has two output files, both of which use XML.

5.2.4.1 IntermediateSave.xml

The first output file is "IntermediateSave.xml", where all the information pertaining to the grafcet components is contained along with the newly made associations described in subsection 5.2.3.

The following listing 5.1 illustrates the XML structure of this output file. The tag "<Page>" represents the previously mentioned "pages" with their respective name and identifier. Moreover, it contains the tag "<ComponentCollection>" where all the components belonging to said "page" are listed, which can further have multiple "<Component>" tags. The component element will contain all of the aforementioned component object variables, excluding the spacial and size related ones and including extra variables pertaining to a possible association to a partial grafcet and association to other elements. As described in the previous subsection, not all variables are required for certain components, these element are either populated with a "NULL" value, as is the case for the partial grafcet tags, or with a default value (e.g. a component of type "step" does not need an "edge", its value will be the default "Falling"). This makes the intermediate save file quite similar to those generated by GRAFCET-Studio. This fact does not prove to be problematic, as the second part of the toolchain - section 5.3 - was made to handle this.

Listing 5.1: IntermediateSave.xml XML structure

```
<GrafcetProject>
  <Page>
    <Identifier></Identifier>
    <Name></Name>
    <ComponentCollection>
      <Component>
        <Identifier></Identifier>
        <Name></Name>
        <Type></Type>
        <PartialGrafcetName></PartialGrafcetName>
        <AssociatedMacroStep></AssociatedMacroStep>
        <AssociatedEnclosedStep></AssociatedEnclosedStep>
        <AssociatedPage></AssociatedPage>
        <PreviousComponents></PreviousComponents>
        <NextComponents></NextComponents>
        <IsInitial></IsInitial>
        <TransitionType></TransitionType>
        <Edge></Edge>
        <AssociatedStep></AssociatedStep>
        <Action></Action>
        <Condition></Condition>
      </Component>
    </ComponentCollection>
  </Page>
</GrafcetProject>
```

The following list provides more in-depth commentary regarding each "<Component>" tag field:

- **<Identifier>** - Denotes the identifier given by the GRAFCET-Studio Reader program.
- **<Name>** - Denotes the name given to each component, usually being the type of component followed by how many of said component exist in memory after the addition of this one. For example, the third transition being interpreted by the program would be named "Transition_3".
- **<Type>** - Denotes the type associated to a component, this tag follows the exact nomenclature used by GRAFCET-Studio for its components.
- **<PartialGrafcetName>** - If the component belongs to a partial grafcet of any kind, the tag will display the name given to the group border (the partial grafcet), otherwise if it belongs

to the main grafcet, i.e. no partial grafcet association, it will display "NULL". For example, a step belonging to a partial grafcet name "G1" will have "G1" in this field.

- **<AssociatedMacroStep>** - If the component belongs to a partial grafcet and the partial grafcet type is macrostep expansion, i.e. is associated to the state of a macro step, this field will contain the identifier belonging to the associated macro step, otherwise, regardless of whether the partial grafcet exists or not, if no macro step is associated to the partial grafcet, the value will be "NULL". For example, an action belonging to a partial grafcet, associated to the macro step "Step_M1" with the identifier "5", will have "5" in this field.

- **<AssociatedEnclosedStep>** - This tag works exactly the same way as the "<Associated-MacroStep>" tag; however, instead of the partial grafcet being associated to a macro step, it is associated to an enclosing step.

- **<AssociatedPage>** - Denotes the identifier of the page to which the specific component belongs to. For example, a step belonging to the first read page, i.e. identifier "0", will have the value of this field to be "0".

- **<PreviousComponents>** - Displays the list of all the identifiers of the "previous" components, i.e. components that come right before in a sequence, associated to a specific component. If there are no previous components associated to the component in question, then the tag value will be "NULL". If it is otherwise associated to one or more previous components, their identifiers will be present, separated by a comma between each other. Note that in this last case there will always be a comma present after all the identifiers, even if there is no other identifier that comes after, this proves to be useful for the second half of the toolchain when it comes to reading these values. For example, a step has two transitions that lead to it, one with the identifier "1" and the other "2". The <PreviousComponents> tag value will be "1,2,".

- **<NextComponents>** - This tag works the same exact way as the <PreviousComponents> tag, with the only difference being that instead of handling the list of previous components it handles the following components, i.e. components that appear right after in the sequence. For example, a transition that leads to a single step, of identifier "5". The <NextComponents> tag value will be "5,".

- **<IsInitial>** - Denotes whether a given step has an activation link. Its value is "true" if it has one or "false" otherwise.

- **<TransitionType>** - Exclusive for transitions use, denotes the type of the transition. It takes the value of "Normal" - which also happens to be its default value - for transitions that can have previous and next components associated.It takes the value of "Sink" for transitions that can only have previous components associated. It takes the value of "Source" for transitions that can only have next components associated.

- **<Edge>** - Exclusive for activation/deactivation stored action use, denoting the edge type of said actions. If the action is a stored action triggered on step activation, rising edge, the value of the tag is "Rising", otherwise, if the action in question is a stored action triggered on step deactivation, falling edge, the value of the tag will be "Falling". The default value, applicable to any other type of component, for this tag is "Falling".
- **<AssociatedStep>** - Exclusive for all action and forcing order use, where it denotes the step identifier to which the action is associated to. For example, a continuous action is associated to a step of identifier "12", the tag value will also be "12".
- **<Action>** - Also exclusive for all action and forcing order use, denotes the action content to be performed by said action component, represented by a string. For example, a stored action (irrespective of edge type) that increases a variable "A"'s value by two, will have the tag value set to an expression similar to "A:=A+2".
- **<Condition>** - Exclusive for continuous action with condition and event-driven stored action use, where the tag value denotes the condition that these actions must clear in order for them to perform their expression present in the tag "<Action>". For example, a continuous action with the condition ("<Condition>" value) "B>2" and action "C", must first ensure, besides the step associated to it being active, that the value of "B" is higher than "2" so that it may set the variable "C" to true.

5.2.4.2 SymbolSave.xml

The second output file is "SymbolSave.xml", where all the relevant data from the previously processed symbols is formatted and written.

Akin to the first output file, the XML structure of this output file is represented in listing 5.2. The symbols are related to the tag "<Symbol>", which further contains the respective name, symbol type and the data type.

Listing 5.2: SymbolSave.xml XML structure

```
<SymbolList>
  <Symbol>
    <Name></Name>
    <SymType></SymType>
    <DataType></DataType>
  </Symbol>
</SymbolList>
```

The following list provides more in-depth commentary regarding each "<Symbol>" tag field:

- **<Name>** - Denotes the name given to a specific symbol. For example, "ConveyorA_ON".

- **<SymType>** - Denotes the type of the symbol, which may take the values of "Input", "Output" and "Data". For example, a symbol associated to a sensor will, generally, have the "<SymType>" set to "Input".
- **<DataType>** - Denotes the type of the data handled by a symbol, whose value may range from: "BOOL", "INT16", "INT32", "UINT8", "UINT16", "UINT32" or "FLOAT". For example, a symbol associated to an ON/OFF indicator light will likely have the "<DataType>" value set to "BOOL".

5.3 Structured Text Translator

Similarly to the previous section, this section outlines the objectives of the second component of the toolchain. It then presents an overview of the libraries employed, followed by a description of the program's workflow and how these libraries are integrated within it. As in the previous component, the workflow is divided into several processing phases, with additional subsections dedicated to explaining the key concepts and components involved in certain phases. Finally, the output files generated by this component and their significance are examined.

5.3.1 Objectives

The objectives undertaken by this part of the toolchain involve reading and parsing the two files previously created by the first part of the toolchain - GRAFCET-Studio Reader - and adapting it to the internal tool's model, which is adjacent to the meta model created in chapter 4.

It additionally provides minor internal model checking against the meta model to ensure some basic correctness, not semantic checking, but rather whether the extraction from the "IntermediateSave.xml" file and subsequent conversion into the meta model structure was successful - this internal model is then written and saved to an XML based file - "MetaModel.xml".

Finally, the information in the internal model is used in the assembling, following the proper template files, of the final PLCOpenXML project file, according to the defined transformation rules.

5.3.2 Libraries

For this tool six libraries were used in total. Half of them were borrowed and the other half were created. One of these libraries - **pugiXML** - was already described in the library subsection for the GRAFCET-Studio Reader, as such it will not be further mentioned here.

When it comes to the borrowed libraries, the first is **"inja"**, created by Berscheid, described as "a template engine for modern C++" which allows for the creation of an internal JSON-based data model, dependent on the second library from the ones which were borrowed, that can later be rendered in a multitude of outputs types.

The second borrowed library is **"json"**, created by Niels Lohmann. This library allows for the integration of JSON data structures trees in C++ code prioritizing an “intuitive syntax”, “trivial integration” and test case thoroughness over “memory efficiency” and “speed”.

Regarding the three libraries created to aid the tool, the first was **"aux_funcs"**, where a set of functions are dedicated to ensure the aforementioned meta model correctness. Additionally, there are functions which are used for constructing the "MetaModel.xml" file, making use of the pugiXML to properly format the grafcet components into XML.

Secondly, there is the **"aux_structs"** library, where the structs, enumerations and functions responsible for creating the meta model internal objects are present, as well as some auxiliary functions. Moreover, it makes use of maps that transpose relevant element information from string to an enumeration type and vice-versa. This function works much the same to the one described for the previous part of the toolchain, differing mostly in the model structure used.

Finally, the **"aux_trans_funcs"** library is used throughout the whole translation process of this tool. It consists of all the required functions and variables to build the translation JSON data model, where all the relevant information for the IEC 61131-3 project is contained. It makes heavy use of the JSON library by Niels Lohmann.

5.3.3 Program Flow

Similarly to the previous tool, this section is dedicated to giving more insight into the flow of this part of the toolchain, expanding upon what was mentioned in subsection 5.1.2. Figure 5.3 shows the flowchart created to illustrate this progression.

- **Extraction Phase** - Acquisition of the grafcet information.
  - The extraction phase begins by acquiring all of the information present in the intermediate file created by the previous tool.
  - Afterwards, the function **create_element()** is utilized to populate the internal model with the newly read file information, saving them and following elements onto an object model based on the meta model from the previous chapter.
- **Inter-Component Association Phase** - Make component-component associations via the object oriented model and create connected/partial grafcet elements.
  - After this extraction phase comes the inter-component association phase, beginning with the function. **make_arc_associations()**. This function goes through all components and their previous/next component lists and makes the appropriate associations via pointers to and from other components. Additionally, it handles the associations made between steps and actions, associating the latter to the former.
  - After the associations are made, the **create_grafcets()** function is called, performing two main tasks. First, it creates all connected grafcet objects by checking the component-component associations, populating them with all the elements that belong

to it. Second, it creates all the partial grafcet objects, checking whether the aforementioned connected grafcets belong to any given one, adding them to the partial grafcet's connected grafcet list if true.

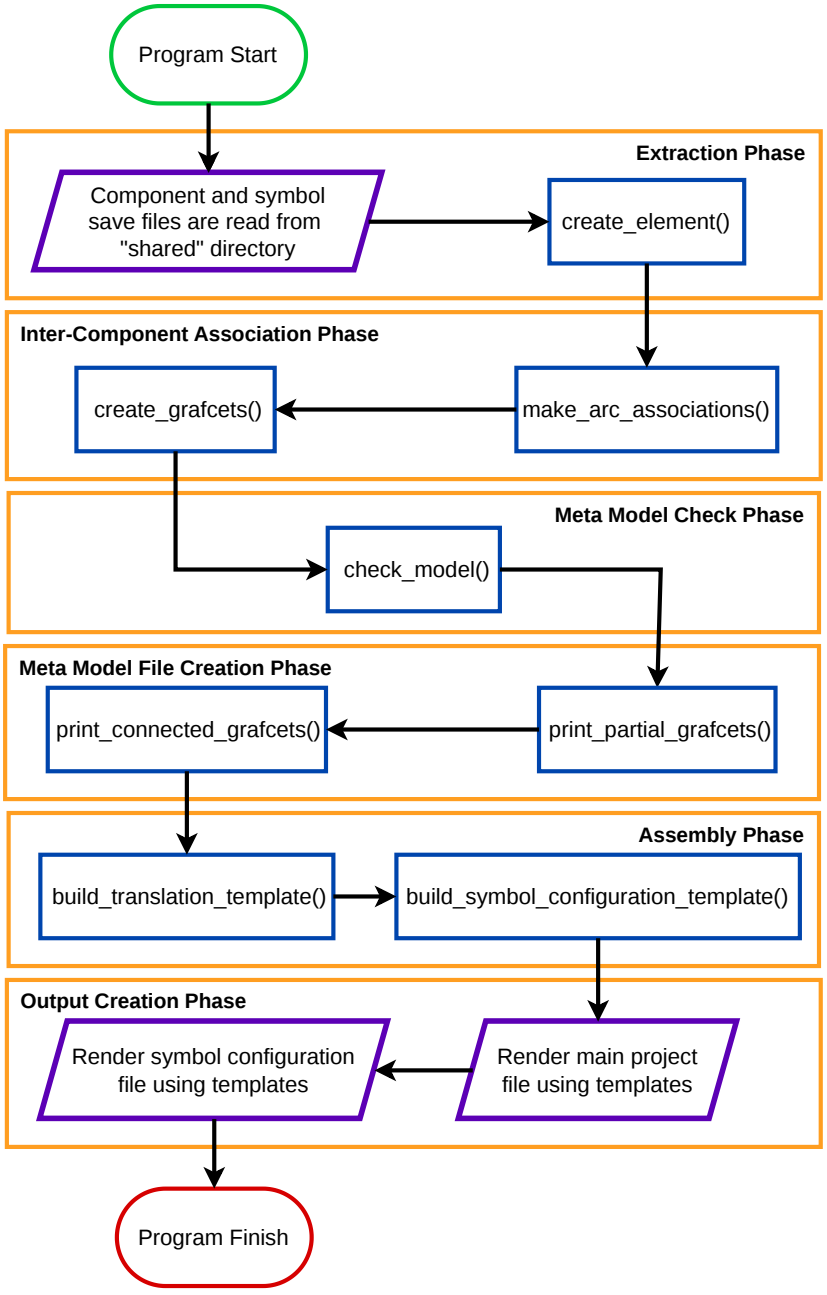


Figure 5.3: Flowchart representing Structured Text Translators' program flow.

- **Meta Model Check Phase** - Ensured correctness of the internal model.

  - In order to ensure that the created internal model is congruent with the information extracted from the intermediate save file, the function **check_model()** is used. The function iterates though all partial grafcets, connected grafcets and their subsequent components, checking whether the information contained in their variables matches that of the save file entries and, additionally, whether the information makes sense in the context of the variable it belongs to. At the end of this phase, a console print indicates if an object's status is correct or compromised.

- **Meta Model File Creation Phase** - Creates the file containing the information from the internal model.

  - Following this analysis, comes the file creation phase dedicated to the meta model. It begins by handling the information for the partial grafcet via the **print_partial_grafcets()** function, where the information about the partial grafcet, their respective connected grafcets, and the components of each of the connected grafcets is formatted.
  - Then comes the **print_connected_grafcets()** function, which saves the information regarding all of the "loose" connected grafcets and their associated components. The term "loose" is used here to denote that the connected grafcets belong to the main grafcet and not to a partial grafcet. Finally, the file formatted via these two functions is saved to the "shared" directory.

- **Assembly Phase** - Translates the the grafcet components into IEC 61131-3 constructs, whose information is saved in a JSON (appendix C) format for use with the templates of the next phase.

  - This phase describes the translation of the grafcet concepts into the IEC 61131-3 constructs, a process which begins with the function **build_translation_template()**. Besides handling metadata, the function begins by initializing the POUs and orders them according to their hierarchical relations. Then the translation itself begins, mostly following the GRAFCET translation rules (appendix D) established by [33]. Multiple other functions are used for this effect they are, by order of execution:
    * **build_transitions_template()** - Handles the rules pertaining to the transitions, making use of the function **parse_conditional_statement()** to analyze a transition's condition.
    * **build_steps_template()** - Applies the step related translation rules.
    * **build_actions_template()** - Applies the translation rules for the actions. Depending on the type of action other functions are called. If an action is continuous with a condition it calls **parse_conditional_statement()**, if it is stored it calls **parse_stored_action()** and if it is a forcing order it calls **parse_forcing_action()**.

    Additionally, other functions are utilized to handle certain CODESYS constructs, such as triggers and time conditions.

  - The second function used, **build_symbol_configuration_template()**, handles the symbol data formatting. This process additionally considers their access type when used with OPC-UA, which can be read-only, write-only or read/write.

- **Output Creation Phase** - Applies the previous translations to template files, creating the output files.

  - Regarding the output creation phase, the first file which is handled is the main project file, containing all the relevant project information, such as the translated constructs, task information and the main POU program.
  - The second and last file contemplates the CODESYS specific component, Symbol Configuration, where all the symbols are displayed for OPC-UA use.

5.3.4 Outputs

The Structured Text Translator has two output files, with one written in PLCOpenXML and the second one in plain XML - file example listings are displayed in appendix B.

5.3.4.1 CODESYSProject.xml

The first file, "CODESYSProject.xml", is the main project file, where all the information regarding the POU structuring, their methods, the variables and the translation content are contained. Since the background section on PLCOpenXML already addresses the overall file structure, section 2.4, this one will focus on providing a more in depth look at how the templates provided in the "template" directory help manifest the final file with the help of the JSON model.Note that file sections considered irrelevant will not be presented, as they do not have a major impact on the fundamental understanding of the file.

Within the "Application" object of CODESYS, which itself belongs to a "Device" object, is contained all the relevant file information, starting with the tasks.

The task structure, illustrated in listing B.1, makes use of very few JSON entries, mostly task related such as the task name, "{{ task.name }}", and its UUID, "{{ task.UUID }}". Additionally it references the name of the POU whose instance will be performed as per the task specifications - "{{ plcprg.name }}".

After this task related section, comes the global variables section - listing B.2 - with the tag "<globalVars>", identifying all global variable lists with their name, "{{ gvl.name }}", which in this case will be only one. This is followed by the first two for-loop structures in the file, where the first writes all the function block variables contained in "gvl" > "variable_list" > "function_blocks", whilst the second writes the remaining variables in the variables list from the path "gvl" > "variable_list" > "others". Finally, an "addData" element completes this section with the writing of the GVL's UUID with "{{ gvl.UUID }}".

Then comes the data section dedicated to the main POU, the "PLC_PRG".

This section - listing B.3 - begins by writing its name in "{{ plcprg.name }}", followed by the declaration of the POU specific variables within the tag "<interface>". For our purposes only local variables were used here, and since in this particular case these do not change from program to program, they were coded into the template file. These variables are "globalStableState" initialized as "false", which keeps track of the inner loop stability (whether a situation is transient or not), and "startProgram" initialized as "true", denoting whether it is the first main loop iteration of the program since its start.

The following "PLC_PRG" related code portion is further subdivided into 6 sections, where the "body" of the POU is addressed. This "body" refers to the code part that will be executed during runtime according to the task configurations and the aforementioned GRAFCET interpretation algorithm.

It starts off with the trigger handling methods, where a for-loop writes all the necessary function block methods, if there are any, from all function blocks using the JSON structure "plcprg" > "body" > "fb_triggerHandler" - listing B.4.

Following this, the time condition check methods are written using the path "plcprg" > "body" > "fb_timedConditionHandler" and also making use a for-loop to write the method for all function blocks - listing B.5.

After which comes the structure that runs only once at the first cycle of the program, using "startProgram" for this - listing B.6. Within the dedicated if-check structure, the template will contain the step trigger handler methods and the stored action handler methods for all function blocks that have them. The stored action methods are only printed after all the step trigger methods.

This structure is then followed by the section for checking whether a transition is ready to be fired or not. This is achieved by writing all the methods from "plcprg" > "body" > "fb_evaluateTransitions" for each POU - listing B.7.

Then comes the biggest partition of the "PLC_PRG" part of the template - listing B.8 - where the stability loop is located. This while-loop begins by checking whether a stable state has been reached via the "globalStableState" variable, always running at least once, if it has then the loop will finish, otherwise it will continue until a stable state is reached. The first methods used are found in "plcprg" > "body" > "fb_evaluateFireableTransitions" and they check the fireable condition and any of the associated previous steps are active, firing the transition if true. Up next is the state evolution with methods in "plcprg" > "body" > "fb_stateEvolutionHandler", where the steps are activated/deactivated accordingly and the stable state is evaluated (any change here results in an unstable state), followed by the activation of macro/enclosing step expansions using the "activateExpansions" methods from "plcprg" > "body" > "fb_activateExpansions". Finally, this partition comes to a close with the exact same code block for step triggers and stored actions as the one presented in listing B.6 without the if structure.

The final part of the main POU's "body" - listing B.9 - handles the pull-down signals, a feature added by [33] to ensure all the necessary continuous actions variables are returned to their

initial state (false), using the methods in "plcprg" > "body" > "fb_pullDownSignals". Moreover, the continuous actions, now considering the step changes that transpired in the previous while-loop, are then activated accordingly, resorting to the methods from "plcprg" > "body" > "fb_continuousActions".

Finally, the last "PLC_PRG" part - listing B.10 - simply addresses the attribution of its UUID via the template variable "{{ plcprg.UUID }}".

The next structure to consider addresses all the remaining POU's - listing B.11 - those that contain the information in the grafcets, be it the main grafcet or any other partial grafcet. Starting with a for-loop that iterates through "pou_list" so that all POU data is present; it goes on to attributing the name of the given POU with "{{ pou.name }}". Then its local variables are considered, writing all the miscellaneous variables ("pou" > "misc_variable_list") and the transition variables ("pou" > "transition_variable_list").

Following this is the part - listing B.12 - where, for a given POU, all of its method information is written from "pou" > "method_list". For all methods, it begins with the name, "{{ method.name }}", and UUID, "{{ method.UUID }}". Addressing afterwards the interface, where not only do all method variables belong ("method.variable_list"), but the return type also appears, "{{ method.return_type }}". This section ends with the code for the method in question, the "body", where all the string lines from its vector ("method.body") are written sequentially.

Finally, the per POU UUID is added with "{{ pou.UUID }}" - listing B.13.

5.3.4.2 SymbolConfiguration.xml

The second file, "SymbolConfiguration.xml", addresses the symbol configuration, containing information regarding UUIDs of multiple, irrelevant, components (as such not considered here), information about the symbols themselves and to which CODESYS construct they belong to. Unlike the previous file, this one is far more CODESYS oriented, since the existence and use of components akin to a "symbol configuration" is not standardized in the same way PLCOpenXML is - the mere presence of this file indicates as much, not being a part of the previous file ("CODESYSProject.xml") capable of being integrated with the PLCOpenXML standard. However, similarly to the previous section, subsubsection 5.3.4.1, this section shall only provide information regarding the template and its JSON model integration for the sake of simplicity.

Regarding this template - listing B.14 - if there are any symbol containing entries present in the vector "{{ symConfig.entries }}", these will be iterated upon in the for-loop. This loop begins by writing the data relevant to the container where the symbols are found, which happens to only be its name, "{{ ent.name }}". Following this is the actual symbol data, by simply adding all vector ("ent.variable_list") information for each index, thus ending the template save writing. Note that the algorithm that builds the JSON model used in this section, i.e. the build_symbol_configuration_template() function, built the symbol data before storing it at a given vector index, since they were trivial compared to the rest of the template.

5.4 CODESYS Import Script

This section presents the import script, written in Python, developed for use with CODESYS, beginning with an outline of its objectives, followed by a brief description of the libraries used and the CODESYS Application Programming Interface (API). It then concludes with a description of the script's implementation and operation.

5.4.1 Objectives

The main objective of this script is to simply provide a way for the aforementioned toolchain parts, GRAFCET-Studio Reader in section 5.2 and Structured Text Translator in section 5.3, to be executed, transforming a given GRAFCET save file into the proper outputs outlined in subsection 5.3.4, and for this final information to be loaded onto the CODESYS project environment, bypassing any tedious manual import work.

5.4.2 Libraries and API

This import script makes use of two Python libraries, "os" and "subprocess".

The library "os" is used for a multitude of function; however, for this script's purposes it only serves as a way to access the files in a given directory path.

Whilst the library "subprocess" is, for the purposes of this script, used simply to "call", i.e. execute, programs/scripts.

Furthermore, the API to which the section title makes reference to is the CODESYS API. As any other API, it allows for a faster, better and more secure access to the CODESYS internal processes, greatly facilitating the creation of automation scripts such as this one. Owing to the complexity presented by said API, the fact it is not the focus of this work and that it was sparsely utilized, means it will not be addressed in any more detail throughout the following sections than here.

5.4.3 Program Flow

This section highlights this import script's flow from beginning to end, reviewing its three main phases.

Similarly to the outputs section of the Structured Text Translator - demonstrated in appendix B - this section also makes use of the expression "(...)" in its listings to indicate that text has been omitted or to indicate continuity with another previous/following listing.

In order for the script to function correctly it must be executed from within the CODESYS environment via the appropriate "run script" option in the toolbar and, preferably, on an empty project file .

Further note that, throughout this section, files are imported a number of times into CODESYS, for which a certain CODESYS project component must be selected that will "receive" this imported information. This is expected, as it stems from the fundamental fact that CODESYS components are hierarchical, where a given component cannot belong to another one directly, e.g. a POU cannot be assigned directly to a project it must be associated to an application, which follows the hierarchical relations of: application < PLC logic < device < project.

5.4.3.1 Information Gathering Phase

The script begins with the code on the listing 5.3, where it begins by declaring the paths to the processes used further down, acquiring the path for both of the previous developed programs and the eventual path for the two final files used to import both the CODESYS project ("CODESYSProject.xml") and the symbol configuration ("SymbolConfiguration.xml"), making use of the "os" library for this. Furthermore, it ensures that these paths are represented relative to the script and are not absolute paths, otherwise it could cause problems for future installations with different directories.

Afterwards, the script proceeds to execute the two previous C++ programs in the proper order with the aid of the "subprocess" library, starting with the Grafcet-Studio Reader followed by the ST Translator. Now, the previous two file related paths no longer point to non-existing files as they have just been created by the latter program.

Listing 5.3: CODESYS Import Script - Part 1

```
(...)
# Path declarations
script_dir = os.path.dirname(os.path.abspath(__file__))
grafcet_src = os.path.join(script_dir, "GRAFCET-Studio_Reader",
    "src")
st_translator_src = os.path.join(script_dir, "ST_Translator", "
    src")
plcopenxml_path = os.path.join(script_dir, "shared", "
    CODESYSProject.xml")
xml_path = os.path.join(script_dir, "shared", "
    SymbolConfiguration.xml")
#####

# Execute relevant toolchain programs
print("[INFO] Executing Grafcet-Studio Reader...")
subprocess.call(os.path.join(grafcet_src, "main.exe"), cwd=
    grafcet_src)
print("[INFO] Grafcet-Studio Reader finished.")
```

```
print("[INFO] Executing ST Translator...")
subprocess.call(os.path.join(st_translator_src, "main.exe"), cwd
    =st_translator_src)
print("[INFO] ST Translator finished.")
#####
(...)
```

5.4.3.2 Main Project Import Phase

Following this is the phase where the main project, "CODESYSProject.xml", is imported.

It begins by attempting to find the project, i.e. the base component of a CODESYS project even when it is completely empty, here referred to as "projects.primary".

After this it simply imports the GRAFCET translation information from the save file with the project function "import_xml()" (used for PLCOpenXML files). The script utilizes the CODESYS API for both of these actions.

Listing 5.4: CODESYS Import Script - Part 2

```
(...)
# Find project
proj = projects.primary
if proj is None:
    raise Exception("No active CODESYS project.")
#####

# Import PLCOpenXML project
print("[INFO] Starting main project import...")
proj.import_xml(plcopenxml_path, proj)
print("[INFO] Finished main project import.")
#####
(...)
```

5.4.3.3 Symbol Configuration Import Phase

This last phase begins by trying to acquire the application component, which was just created via the previously imported file, by searching in "proj.active_application".

After which, the "SymbolConfiguration.xml" file is imported onto this component with the "import_native()" function (used for other XML files which are not PLCOpenXML).

Finally, the project is built using the "build" function. This is done to ensure that the symbol configuration variable list is updated, given that by importing this configuration CODESYS considers it as a change being made to the existing variables.

Similarly to the previous phase, the CODESYS API is invoked thrice.

Listing 5.5: CODESYS Import Script - Part 3

```
(...)
# Find created application
app = proj.active_application
if app is None:
    raise Exception("No active application in the project.")
#####

# Import symbol configuration XML
print("[INFO] Starting symbol configuration import...")
app.import_native(xml_path)
print("[INFO] Finished symbol configuration import.")
#####

# Build project (mostly because of symbol configuration)
print("[INFO] Building...")
app.build()
print("[INFO] Finished building.")
#####
(...)
```

5.5 Summary

The purpose behind this chapter was to give an explanation regarding the toolchain implemented by this work.

It begins by analyzing the overall layout of the toolchain: how is its directory structured, what are the programs and files in said directory and what is the program workflow.

Following this, the first part of the toolchain was presented, GRAFCET-Studio Reader, where the grafcets created with GRAFCET-Studio are parsed via their save files, processed so that the inter-component connections are made and then saved onto a file, "IntermediateSave.xml", as are the symbols acquired, "SymbolSave.xml", for further processing by the next part of the toolchain.

This "further processing" is handled by the Structured Text Translator, which, after parsing the information from "IntermediateSave.xml" according to the meta model developed, processes its component relations further by creating grafcets (partial and connected) and associating them to said components, performing a meta model analysis to ensure correctness in the internal model. Moreover, the program then runs the translation algorithms and the symbol-related algorithms (after parsing "SymbolSave.xml"), saving this final information in a JSON format for it to then be used alongside template files to create the final project outputs - "CODESYSProject.xml" and "SymbolConfiguration.xml".

Finally, the CODESYS Import Script makes use of the CODESYS API to execute the previously described tools of the toolchain in sequence, thereby generating the two final output files and importing them into an empty CODESYS project. The script is executed from within CODESYS to ensure that the API is accessible and that the import paths are correctly resolved.

Given this, the following chapter will illustrate the tests performed to ensure the toolchain's correctness in its translation objective.

Chapter 6

Project Validation

This chapter presents all the relevant information regarding the toolchain validation done to ensure that the results provided by the algorithms developed, mentioned in the previous chapter 5, correctly reflect the created grafcets.

It begins by presenting the overall objectives that these tests seek to accomplish. Followed by presenting a description of the process undergone for the grafcet testing.

Afterwards, the GRAFCET tests themselves are addressed, first presenting the particular objectives the test accomplishes, then a description of the grafcet with an accompanying illustrative figure, for each test.

The next section goes over the tests for OPC-UA. It starts by showing the objectives fulfilled by such tests, followed by describing the test process and context. Then a description of one of the multiple tests performed is demonstrated, with auxiliary figures.

Lastly, a summary of the chapter is presented.

6.1 Objectives

The tests performed on the toolchain encompass a multitude of objectives, among them are:

- **GRAFCET Correctness** - Extensive testing ensures that the semantics behind the original GRAFCET charts are properly reflected on the translated version. In other words, if both implementations are under the same input circumstances and step situation, then the next cycle iteration will yield the same outputs and state changes, for every cycle.

- **Translation Rule Correctness** - Although a given GRAFCET translation may be correct as per the previous entry, it may still suffer from incorrect rule implementation. These rules, which refer to the GRAFCET interpretation rules outlined in a previous chapter, may not be correctly implemented in the final project as defined by the translation rules. This may not cause any problems for a smaller sample of cycles; however, given a large enough cycle sample, small, nigh imperceptible errors may start to accumulate from improper method execution, with the possibility of having adverse effects during runtime, from a larger error

manifesting or a simple program crash. Testing and comparison to the established rules aims to curb this issue.

- **Variable List Congruity** - The final objective to consider is whether the variables implemented match up to the model they were taken from - an objective quite similar to the previous one but simply applied specifically to variables. By testing and checking the resulting variables from the toolchain with the ones from the model, inconsistencies such as unused variables, missing variables, incorrectly applied variables and even improper OPC-UA variable selection can be avoided, preserving the integrity of the final program.

6.2 Testing Process

In order to ensure an adequate test coverage and results, a simple testing process was employed:

1. It started by first outlining the specific elements and types of grafcet chart sequence that should be studied on a test-by-test basis.
2. Given the overall structure idea and the elements to be tested, a test grafcet was created in GRAFCET-Studio.
3. This grafcet is then simulated using the same tool. It is studied to ensure that the structure is correct and that it aligns with what was defined in point 1, adjustments are made if necessary, repeating this step until the final grafcet is achieved.
4. This grafcet save is then processed by the toolchain via the CODESYS script mention in chapter 5.
5. Perform a cursory inspection of the generated code within the CODESYS environment to check for errors or inconsistencies.
6. Execute the CODESYS project implementation and check for further errors.
7. Finally, the symbol configuration variables used for implementing the OPC-UA server were observed, checking whether there were any missing or incorrect. This process was aided by the use of an external OPC client, mentioned in section 6.4, that helped ensure that the server, and subsequently the variables, were operating as expected.

Note that all of the error checks performed in the testing process are done manually, i.e. no external script or program was used to facilitate this process, as it would most likely take more time developing the tools required rather than simply testing directly, with the majority of errors being conspicuous and easily corrected, not warranting such robustness. Furthermore, regarding the error checking from the CODESYS part, these tests were performed side-by-side with the original GRAFCET-Studio implementation. This side-by-side method simply means that a change was applied to the grafcet in GRAFCET-Studio and observed, with the equivalent change being applied

to the CODESYS implementation and also observed, comparing one another, which proved crucial to ensure the logic behind the chart was correct and that there were no inconsistencies or errors.

6.3 GRAFCET Tests

The following eight subsections present the final tests performed, according to section 6.2, to ensure the points outlined in section 6.1.

Each individual test will begin by highlighting the objectives behind its grafcet chart structure, followed by describing said structure.

A validation test is considered to be successful if the CODESYS implementation is capable of performing the exact changes imposed upon the GRAFCET-Studio grafcet, in other words, if the system situation for both is equal given a certain set of inputs, otherwise the test fails. Regarding the test success status, every test presented below has proven to pass successfully, ensuring all the metrics laid out in the objectives, for the most recent version of the toolchain developed. The subsections may additionally remark on other important points, specific to the test, when required.

Moreover, these tests will be accompanied by a figure depicting the according grafcet chart created for it. On the other hand, no information regarding the details of the CODESYS implementation, including figures, will be provided. This decision was made on the basis that this effort would prove quite complicated as the final CODESYS implementation, even when made for a simple grafcet, has too many components. Attempting to document and compare these components with the original grafcet chart would result in a very complex and rapidly expanding test description, something that is was decidedly undesirable for this work.

Additionally, with the objective of making the testing process easier and more streamlined, some of the tests below have a dedicated POU of the type program, named "EXEC_ORDR", which turns certain variables, defined by the user, true or false for a certain amount of time, essentially creating multiple states that cycle through each other in a certain order. This proves useful as, apart from offloading the tedious and repetitive process of manually changing variables states, it also allows for easier and faster testing, specially when trying states where two variables that may cause a transient situation are utilized. On the other hand, some of the tests did not require such a POU, a simply manual variable change proved efficient enough. The decision to use or not use this auxiliary POU was done on a test-by-test basis, applying this POU to projects that would benefit most from it.

6.3.1 Test 1 - Simple Elements

The first test was designed to illustrate the simplest possible grafcet chart sequence, whilst still incorporating the most notable GRAFCET elements: steps, transitions and actions. With this, most grafcets created are sure to function with the toolchain, as they all invariably incorporate a combination of these three base elements.

This test, illustrated in figure 6.1, denotes a initial step, two simple step and three transition looping sequence. Additionally, two continuous actions without condition are employed, one added to the second step "2" and the other to the third step "3".

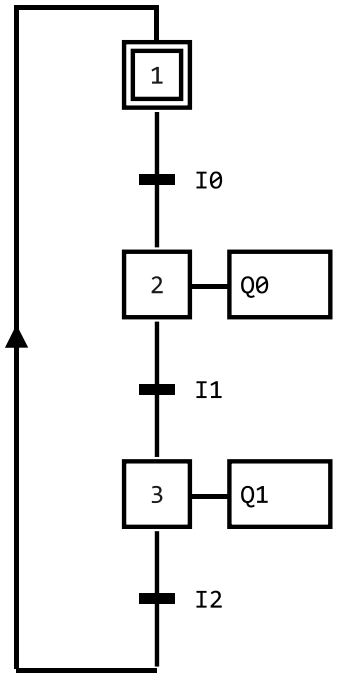


Figure 6.1: Grafcet test chart 1.

6.3.2 Test 2 - Transition Types

The following test, test 2, tackles some additional points relative to test 1. Not only does it test the existence of multiple initial steps, instead of just one, but they also all belong to the same connected grafcet. Another two crucial elements are tested as well, parallelizations and synchronizations with a varying number of inputs and outputs. Apart from the use of normal transitions, the other different types of transition are addressed - source and sink. Lastly, the "selection" type of grafcet structure is utilized, which consequently also helps test the use of horizontal transitions.

The test is shown in figure 6.2, beginning with a source transition to a step. This step is subject to the selection structure, with a transition connecting to the parallelization and the other to another step, an initial step, which itself is connected to the same parallelization via a transition. The parallelization opens to two steps, both of which are of the initial variety, step "4" and "5". Step "4" goes onto connect to a transition, which connects to another step, which connects to the synchronization. Whilst step "5", directly connects to this same synchronization. Finally, the synchronization is subject to the sink transition below it, putting an end to the chart.

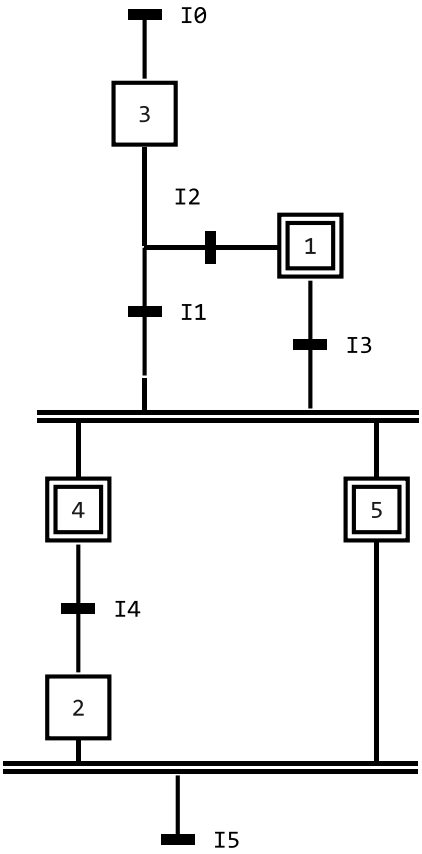


Figure 6.2: Grafcet test chart 2.

6.3.3 Test 3 - Action Types

Test 3 tests the multiple action types, from continuous actions with or without a condition, to stored actions of type on activation/deactivation/event. It also tests how the toolchain handles multiple actions associated to a singular step and, additionally, if the jump element from GRAFCET-Studio is properly incorporated.

Figure 6.3 depicts test 3, having a similar structure to that of test 1, a cyclical sequence, which begins with the initial step "1". Via transition, this step connects to step "2", itself connecting to step "3" via another transition and completing the cycle with a jump back to step "1". All three steps in the test have two actions each:

- **Step "1"** has a on activation stored action and a continuous action without a condition.
- **Step "2"** has a on activation and a on deactivation stored actions.
- **Step "3"** has a continuous action with a condition and a stored action triggered by an event.

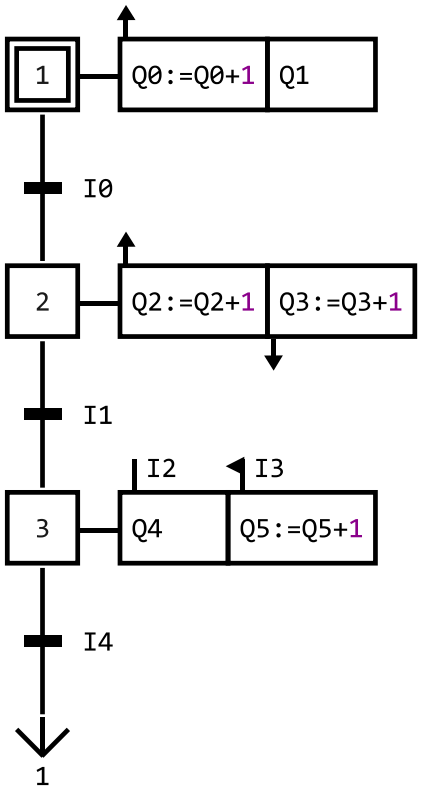


Figure 6.3: Grafcet test chart 3.

6.3.4 Test 4 - Macro Steps

Macro steps and their appropriate expansions were the main focus of test 4. The existence of multiple macro steps and expansions is also tested, alongside the use of actions both associated to macro steps and associated to steps within step expansions. A smaller aspect which was also worth testing was the multiple access to a step from different transitions, inadvertently also testing whether step side connections are working correctly.

The figure 6.4, which illustrates this test, begins with the initial step "1", connected to the macro step "M2" via a transition. This macro step has associated a continuous action with a condition and is connected to the following macro step, "M3", via a transition and jump. "M3" then connects back to "M2" with a transition to its left side, closing the sequence's cycle.

Regarding the macro step expansion related to step "M2", it is named "G1" and begins with a "E1" entry step, connected to the exit step "S1" via a horizontal transition. Additionally, "S1" contains an on deactivation stored action associated, which proved to be an interesting interaction to include into the toolchain.

As for the macro step expansion "G2", associated to the step "M3", it begins with the entry step "E3" connected to a middle step "2" and ending in the exit step "S3". Step "E3" is associated to a on activation stored action, posing a similar situation to that of step "S1" from the previous

partial grafcet. Moreover, step "2" has two actions, a continuous action without a condition and a stored action on activation.

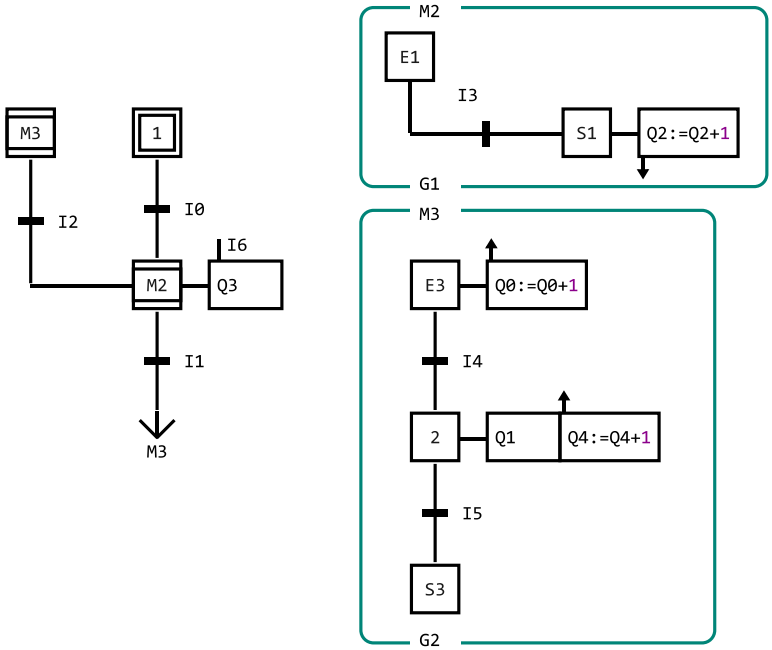


Figure 6.4: Grafcet test chart 4.

6.3.5 Test 5 - Enclosing Steps

The fifth test considers enclosing steps, both normal and initial, and their according expansions. Multiple partial grafcet nesting is also a focus of this test, where the hierarchies and their relations are evaluated. Similarly to the previous test, there are both actions associated to the enclosing steps and actions within the enclosing step expansions.

The grafcet shown in figure 6.5, begins with the initial enclosing step "1"; this step is connected to a "buffer" step "2", i.e. a step which is used solely to test whether the hierarchical implementation described ahead is working correctly and if the initial/activation link dynamic of initial enclosing steps is functioning properly. Both of these steps are connected to each other via two separate transition. Additionally, step "1" contains an on activation stored action.

The enclosing step expansion "G1" is associated to the initial enclosing step just described. Since it is associated to an initial enclosing step, not only does it participate in the initial grafcet situation but it must contain an initial step, in this case step "5", that is only active when the initial situation is considered, either via the first runtime activation or a possible forcing order (not represented here). The other way which this partial grafcet may begin is via the activation link, since in this case the enclosing step is initial it is only after the initial activation has passed that they function. The step to bear the activation link in this test is step "3". The initial step "5" connects to the next enclosing step, step "4", via a transition as does step "3" with a differing transition. All three steps have an action associated:

- **Step "5"** has a simple continuous action without a condition.
- **Step "3"** has an on deactivation stored action.
- **Step "4"** has an event driven stored action.

The enclosing step expansion "G2", associated to the enclosing step "4", contains only a singular step, with an activation link, and a sink transition associated. Additionally, this step has a stored action on activation.

As evident by the enclosing step activation order in the previous descriptions, this chart is beholden to a three layer hierarchy. The first layer, the highest, has the main grafcet associated, the one containing the initial enclosing step. The second layer, the middle, contains the partial grafcet "G1". Finally the third and last layer, the lowest, contains the partial grafcet "G2". As alluded to in the objectives described in the introduction to this section, this hierarchy presents a situation worth testing where the grafcet in a layer below must respect the situation of the ones above it - demonstrating a problem adjacent to that described for the figure 4.2 of chapter 5.

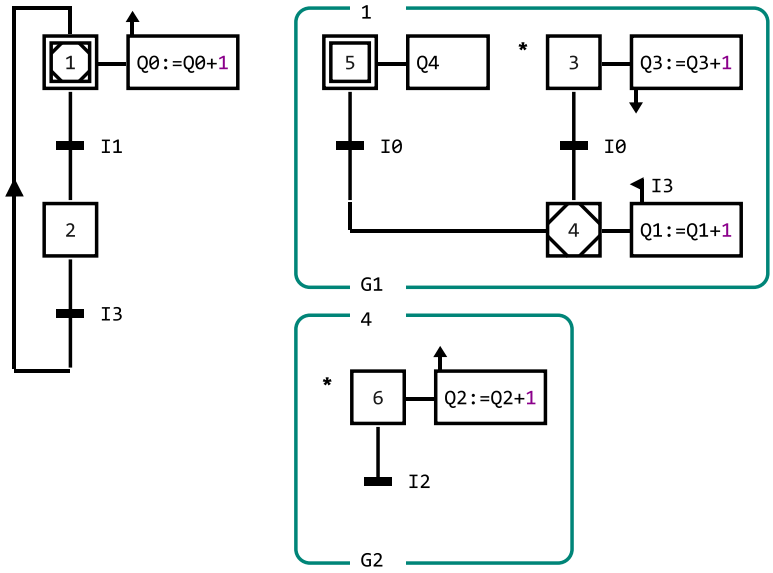


Figure 6.5: Grafcet test chart 5.

6.3.6 Test 6 - Forcing Orders

Test 6 includes the addition of forcing orders, used in conjunction with partial grafcets, whose hierarchical relations will be tested by said orders. Moreover, forcing orders are also used within these partial grafcets. Other actions will also be employed in the partial grafcet to help test how these are affected by the forcing orders.

Since all the grafcets presented in the example participate in the initial situation, this following description will evaluate them by descending order of hierarchy. Note that the grafcets ahead all have a structure of a cyclical sequence. Furthermore, given that the main focus of this test are the forcing orders, only they will be considered when pertinent, described by order of appearance.

Starting with the main grafcet, with the initial step "1", which has no actions or orders. The first order to appear after step "1" is the order of initial situation for partial grafcet "G1", followed by an empty order on the next step and on yet another step a specific situation to activate step "4". Here, after this last step, a "buffer" step is used to test whether the grafcets regain their ability to change situation, given that while forcing order are active this is not possible, and to allow for changes to be made before the next order. After which step "13" has the last forcing order before a sequence cycle is complete, an order to "freeze" the grafcet "G1".

The partial grafcet named "G1" starts with the initial step "3". This grafcet also has forcing orders, containing the exact same forcing order types and in the same order as the previous grafcet. The only notable differences from the previous grafcet to this one are the fact that they act upon the grafcet "G2", the specific situation order is applied to "G2"'s step "6", the "buffer" step from before is gone and actions were added. There is a continuous action without a condition associated to the initial step "3" and a stored action on activation associated to step "7", the same as the empty order of this grafcet.

On the other hand, the partial grafcet "G2", which begins with the initial step "5", does not contain any forcing orders. Instead, its two steps contain one action each:

- **Step "5"** has an event driven stored action.
- **Step "6"** has a stored action triggered on step deactivation.

Given this information, is is clear what the dynamics of each layer of grafcets are. The highest layer, with the main grafcet, controls the middle layer, with grafcet "G1" via its forcing orders, which in turn controls the lowest layer, with grafcet "G2", via their own forcing orders. This results in a test where not only are the effects of standalone forcing orders observed, with the interaction between "G1" and "G2", but the effects of forcing orders upon forcing orders and how these affect the third layer are studied as well, with the interaction between the "main" grafcet, "G1" and, by extension, "G2". The latter of which, similarly to the previous test 5, helps in illustrating the case of figure 4.2 of chapter 5. Proving the test helpful to reinforce the grafcet layer hierarchy by posing a different, easily-changing situation.

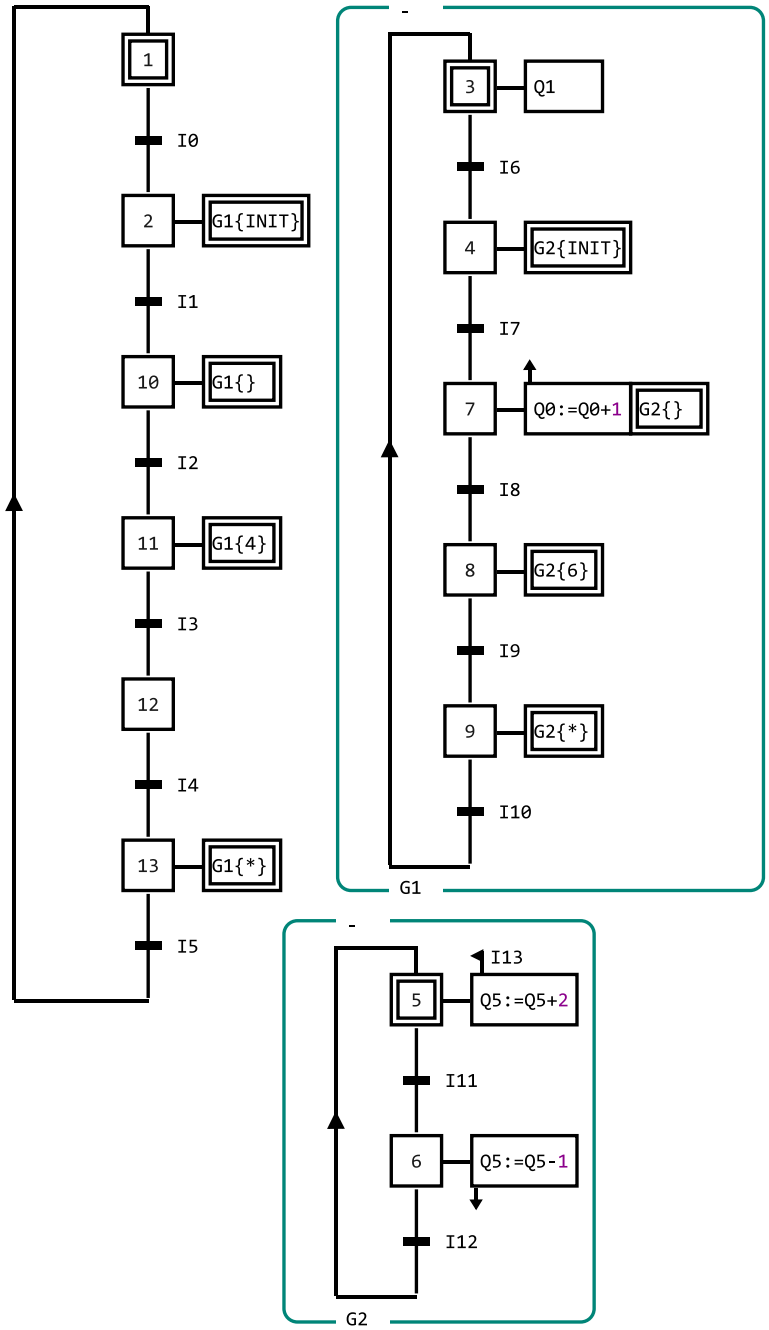


Figure 6.6: Grafcet test chart 6.

6.3.7 Test 7 - Conditions

This following test, test 7, lends itself to an interpretation different from other tests, but somewhat similar to test 6, in that the steps are not the focus of the study. The focal points are the multiple differing conditions presented in the transitions, being its main objective, to test situations where any type of condition may be present.

Similarly to other tests, test 7 also makes use of three grafcets, a "main" grafcet and two partial grafcets, all of them exhibiting the sequence cycle structure - figure 6.7.

The "main" grafcet contemplates five steps and five transitions. The first transition's condition is "↑I0+I1", indicating that either "I1" is true or "I0" goes from false to true. The second transition's condition is "↓I2*I3", indication that both "I3" must be true and "I2" must go from true to false. The third transition's condition is "4s/X3", indicating that step "3", which happens to be the step before this transition, must remain active for at least four seconds for the transition to clear. The fourth transition's condition is "1s/I4/2s", indicating that the input variables "I4" must remain true for at least one second, the latter part of the condition would mean that the condition will still remain true for two seconds after "I4" returns to false; however, given that a condition belonging to a transition needs only to be ready to fire for one cycle, this proves useless. The fifth transition's condition is simply "1", indicating the transition is always fireable.

The partial grafcet "G3" contemplates five steps and five transitions. The first transition's condition is "3s/X6*I5", indicating that three seconds must have passes after step "6" is activated and the variable "I5" must be true, note that step "6" comes before this transition. The second transition's condition is "I7+I6/5s", indicating that either "I7" is true or "I6" is true, "5s" does nothing in this case. The third transition's condition is "X3", indicating that step "3", a step belonging to the previous grafcet must be active. The forth transition's condition is "X13", indicating that step "13", belonging to the next grafcet must be active. The fifth transition's condition is "3s/X10/20s", indicating that step "10", the previous step, must be active for at least three seconds, "20s" is inconsequential.

The last partial grafcet, "G105", contemplates four steps, four transitions and two actions. The first transition's condition is "I9", indicating that the input variable "I9" must be true. The second transition's condition is "[Q0>2+3]+I10", indicating that either the variable "I10" is true or the output variable "Q0" has a value greater than "5". The third transition's condition is "([Q1=4]*I11)+I12", indicating that either the input variable "I12" is true or, the output variable "Q1" has a value equal to "4" and the input variable "I11" is true. The fourth transition's condition is "[Q0-Q1<10]", indicating that the value of the output variable "Q0" minus the value of the output variable "Q1" must be lower than "10". Additionally, both the initial step "11" and the step "12" have, each, an associated event driven stored action.

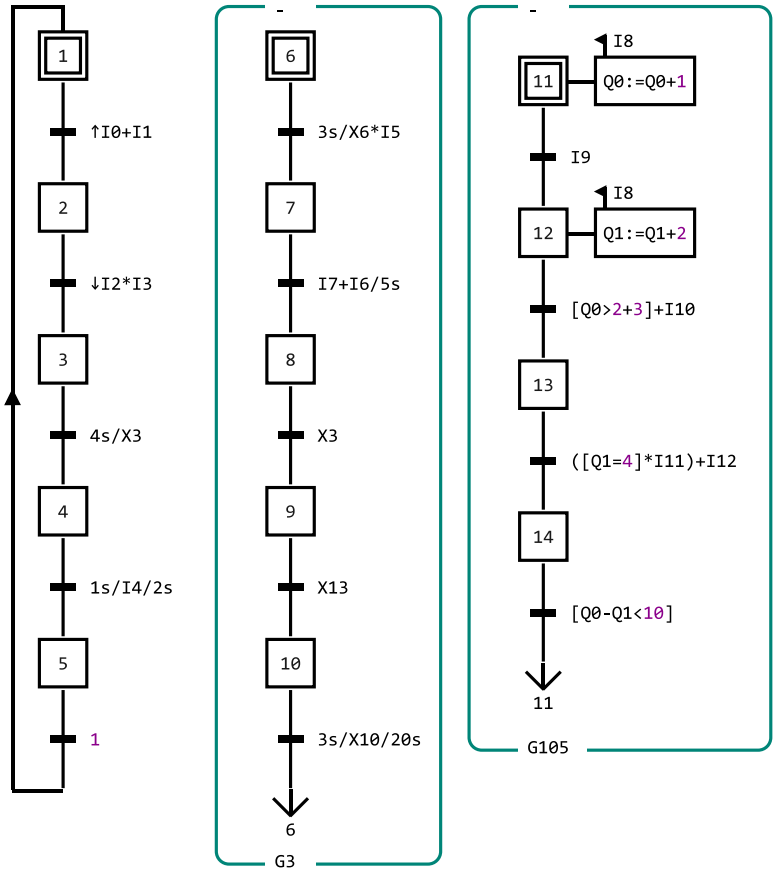


Figure 6.7: Grafcet test chart 7.

6.3.8 Tests 8 and 9 - Realistic Grafcet Implementations

This subsection contains the last two tests performed. Their objective was to test a situation with grafcets that represent a more realistic application scenario where, at least, a majority of the GRAFCET elements and structures were tested. Both of the grafcets test charts presented - figures 6.8 and 6.9 - perform the same function, going about it in different ways.

They were created with the aid of the GRAFCET standard [12], more specifically, the annex B, wherein an example which attempts to mirror a real world scenario is presented in sections B.1 and B.2, illustrated by its figure B.1 [12]. The problem presented is solved by five distinct methods, shown in figure B.2 to B.6 from section B.3 [12]. The grafcets presented for each solution can be subdivided in three main clusters; this fact was used in the following two tests, by combining different clusters into one grafcet chart, new solutions could be obtained but, more importantly, these solutions could test a larger variety of elements per chart, allowing for the creation and testing of only two charts to cover all the necessary elements and structures. Some of the examples shown, namely of figures B.5 and B.6 [12], employ an additional grafcet whose purpose is to act as a "mode" manager, allowing the main structure to be completely turned off and restarted if needed.

Overall, both tests presented make use of normal and initial steps, transitions, continuous actions, stored actions, jumps, parallelizations, synchronizations, selection structures and the aforementioned "mode" manager chart.

On a case by case basis, starting with the test on figure 6.8, its grafcet chart additionally substitutes the bottom element cluster with a macro step and the appropriate macro step expansion. Moreover, the "mode" manager grafcets makes use of forcing orders to manage the main grafcet "G1".

As for the second test, figure 6.9, it makes use of three enclosing step expansion "G2", "G3" and "G4", dependent on enclosing step "5", to substitute the three clusters, controlled by step "3"'s enclosing step expansion chart "G1" via a on/off step state. This latter enclosed step depends on the "main" grafcet which embodies the "mode" manager function, by virtue of being part of it.

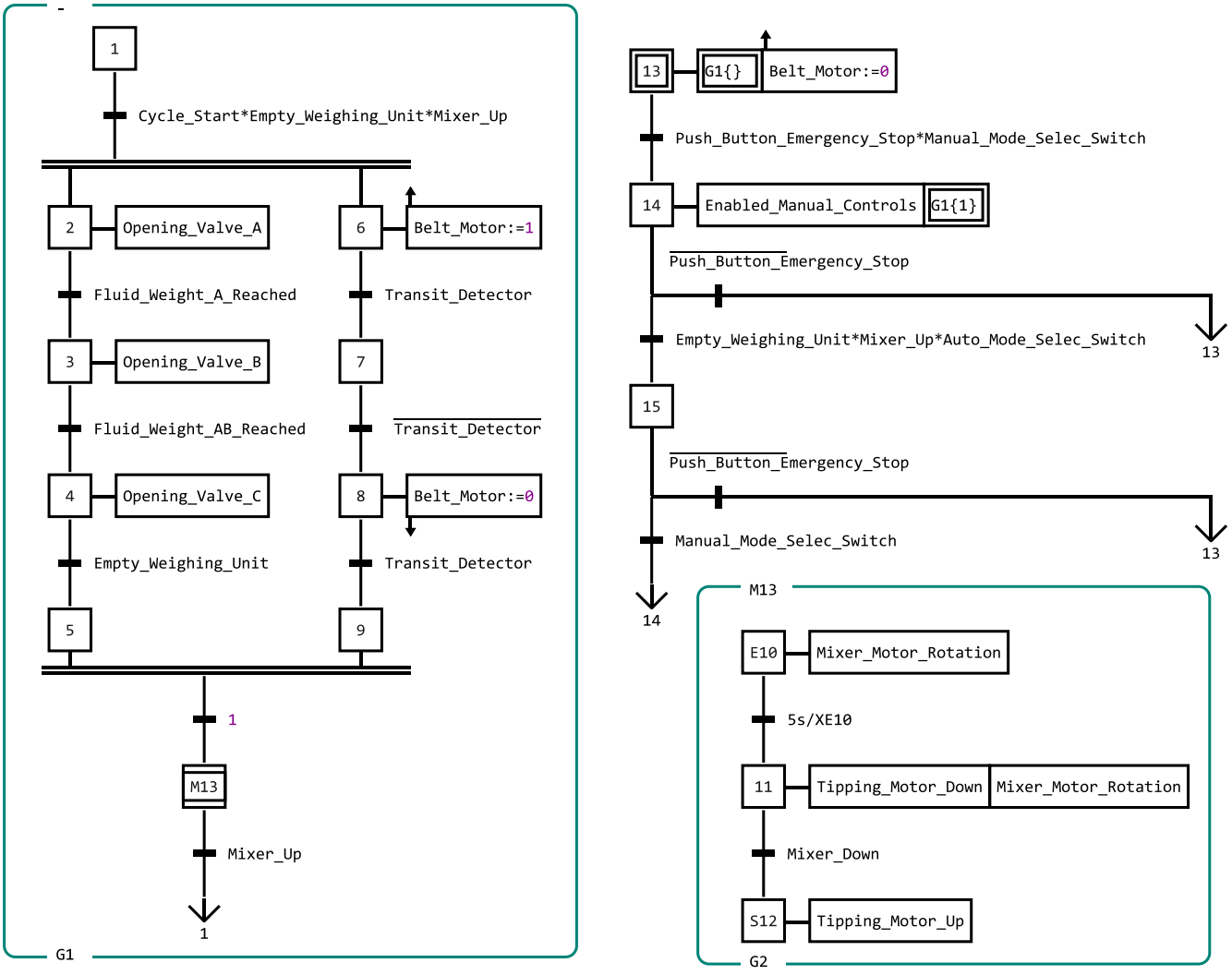


Figure 6.8: Grafcet test chart for the GRAFCET standard, annex B, version 1.

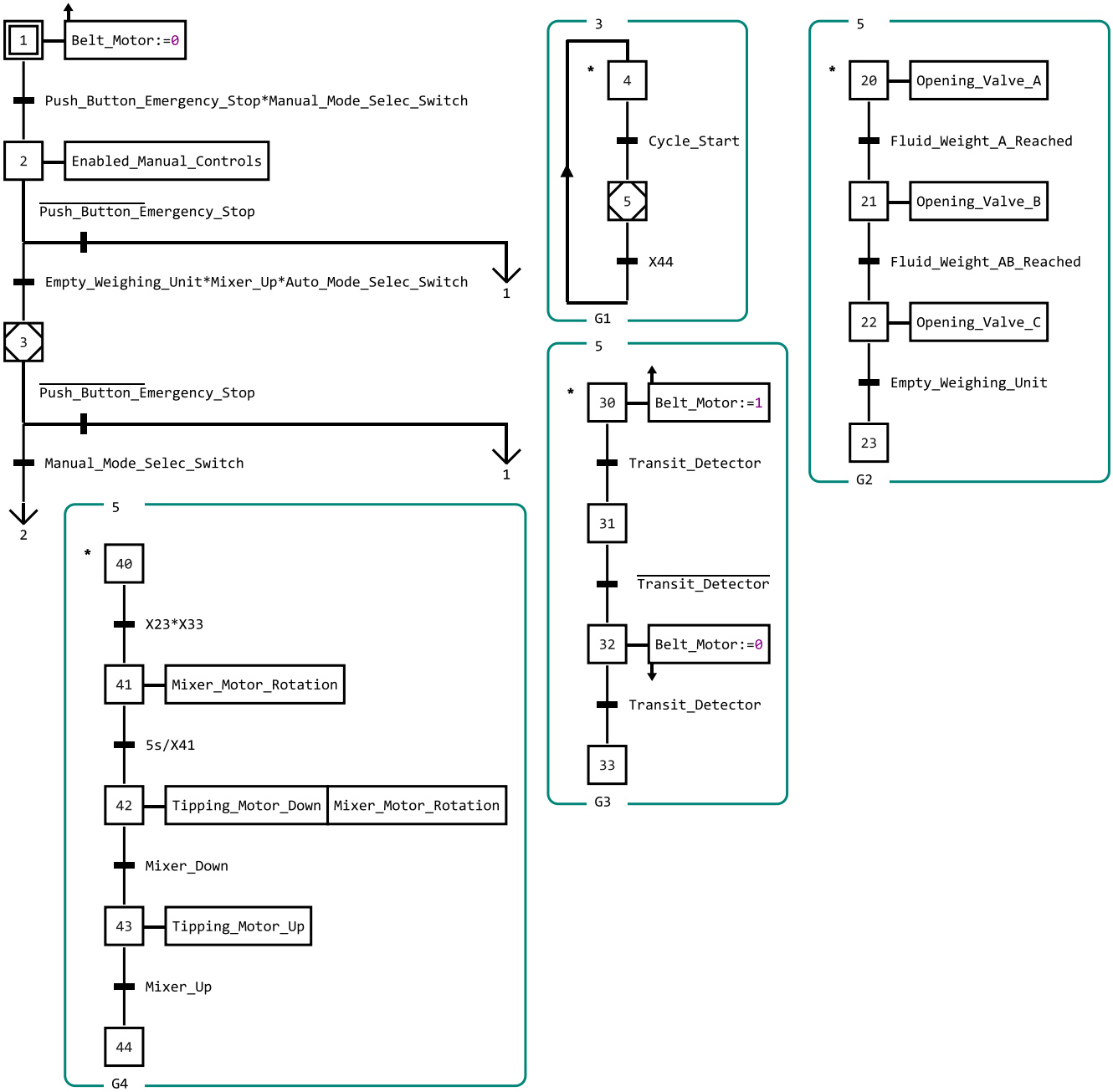


Figure 6.9: Grafcet test chart for the GRAFCET standard, annex B, version 2.

Additionally, both of the IEC 61131-3 projects used in these tests were compared to one another to ensure that they implemented the same logic. This was done by first creating a FACTORY IO scenario that could acomodate both and then sequentially executing the the codes. Their processes were observed with their input situation being altered in the same manner at similar intervals.

6.4 OPC-UA Test

This section describes the OPC-UA test performed in accordance with what is defined in section 6.2. The examples presented in this section only illustrate a singular test case, that of grafcet test 1, as the methodology is kept the same throughout with no notable changes. Note that although this might be the case, all of the tests regarding the symbols and OPC-UA in general were successful.

The variable checks performed for this section of the validation encompass the congruity of said variables, i.e. whether the variables present in the CODESYS implementation are visible to OPC clients, and additionally the variable integrity, where the variable states must be consistent between the server and client and whether the read and write functions for said variables are operating as expected.

For this, apart from CODESYS, a software called "UaExpert" [36], developed by Unified Automation GmbH, was utilized, where an OPC client is setup to acquire the information being exposed from CODESYS, in order to ensure that the symbol configuration built by the toolchain and presented in CODESYS is correct.

For most tests performed, the variables checks were not as thorough like those in section 6.3. Sometimes UaExpert was used where some doubts regarding data integrity and correctness were had during runtime, but generally, symbol checks were only performed at a CODESYS level, with an analysis made on the data presented in the symbol configuration component. This is because, so long as the information in the symbol configuration is correct, the server-client sharing process should be guaranteed. However, if an error manifests even when the symbol configuration seemingly accuses none, it may be indicative of a larger CODESYS OPC-UA server related problem, a topic not in the purview of these tests.

Figure 6.10, demonstrates the symbol configuration created for CODESYS based on the grafcet presented in test 1, where only the variables with a ticked box are being shared via OPC-UA. The follow-up figure, figure 6.11, shows the variables associated to the main grafcet instance "fb_MainGrafcet".

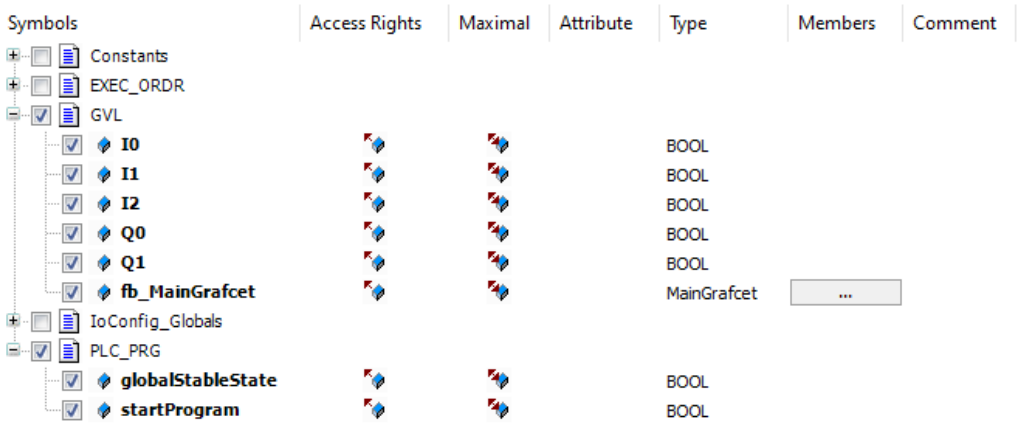


Figure 6.10: Symbol configuration for test 1.

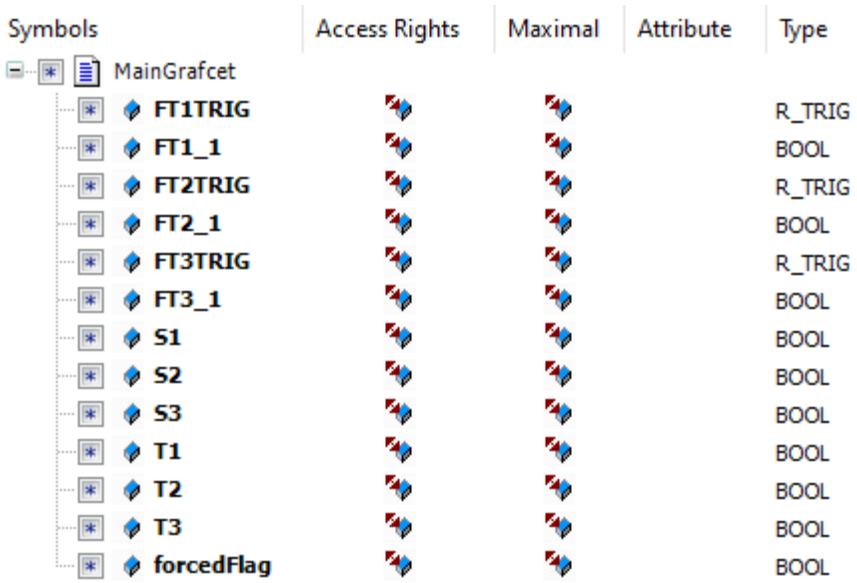


Figure 6.11: Symbols associated to grafcet instance "fb_MainGrafcet".

Regarding the UaExpert environment, figure 6.12 illustrates the "address space" for the symbols presented in figure 6.10 in UaExpert. Additionally, the project structure used in the software is also evident, beginning with "Root" and going down to "Objects" > "DeviceSet" > "CODESYS Control Win V3 x64" > "Resources" > "Application" > "GlobalVars" > "GVL".

Figure 6.13, similarly to figure 6.11, denotes the symbols associated to "fb_MainGrafcet".

Lastly, figure 6.14 demonstrates the "data access view" space, where variables from the previously mentioned address space can be moved into, essentially acting a symbol discriminator, which allows for the information of multiple symbols to be quickly analyzed in large quantities by the user. This proved important as, when the software was used, this window was the main focus, allowing for real-time monitoring of the state changes in any symbol from CODESYS.

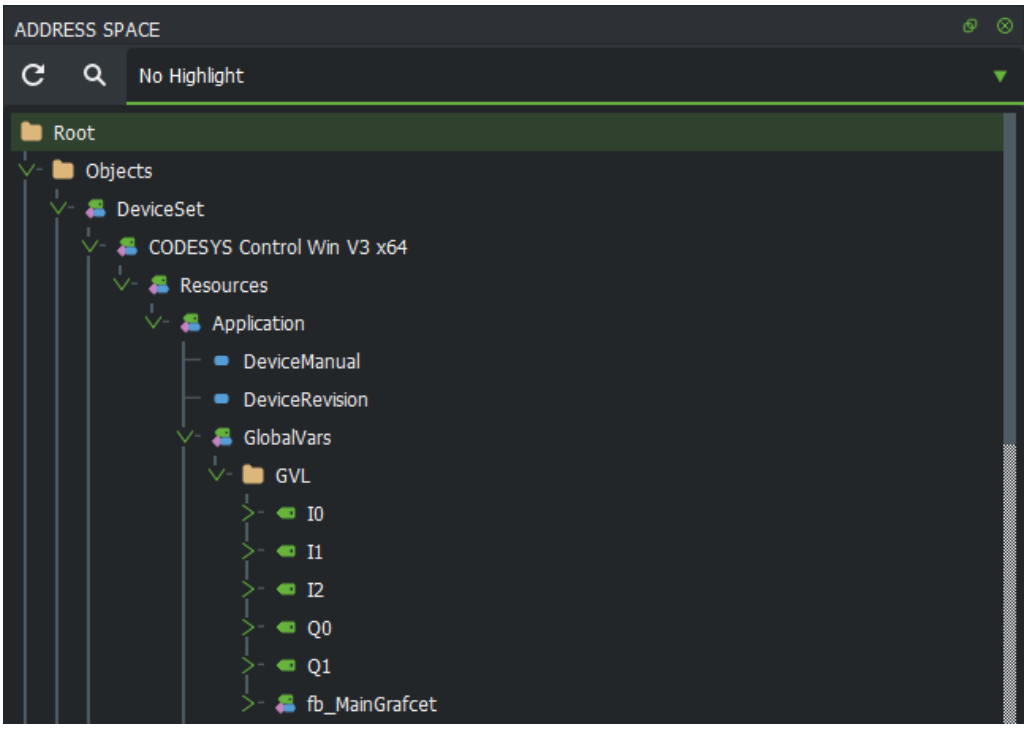


Figure 6.12: UaExpert's address space with symbol configuration symbols.

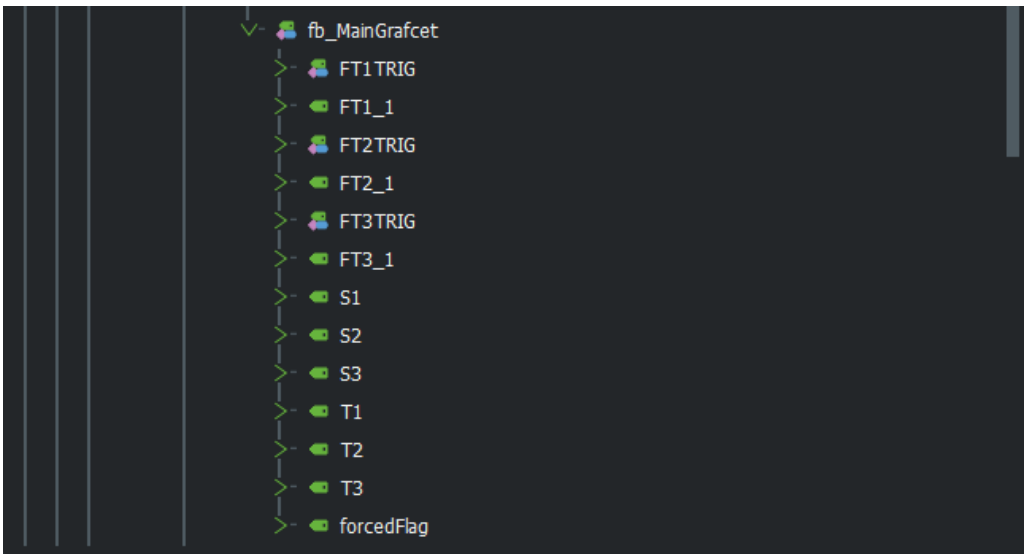


Figure 6.13: UaExpert's symbols associated to grafcet instance "fb_MainGrafcet".

DATA ACCESS VIEW

#	SERVER	NODE ID	NODE PATH	DISPLAY NAME	VALUE	DATATYPE	SOURCE TIMESTAMP	SERVER TIMESTAMP	STAT
1	OPCUAServer@DESKTOP-UFR9O06	NS4|String||var|CODESYS Control Win V3 x64.Application.GVL.I0	GlobalVars/GVL/I0	I0	true	Boolean	23:01:11.036	23:01:11.036	Good
2	OPCUAServer@DESKTOP-UFR9O06	NS4|String||var|CODESYS Control Win V3 x64.Application.GVL.I1	GlobalVars/GVL/I1	I1	false	Boolean	23:01:05.943	23:01:05.943	Good
3	OPCUAServer@DESKTOP-UFR9O06	NS4|String||var|CODESYS Control Win V3 x64.Application.GVL.I2	GlobalVars/GVL/I2	I2	false	Boolean	23:01:10.749	23:01:10.749	Good
4	OPCUAServer@DESKTOP-UFR9O06	NS4|String||var|CODESYS Control Win V3 x64.Application.GVL.Q0	GlobalVars/GVL/Q0	Q0	true	Boolean	23:01:10.803	23:01:10.803	Good
5	OPCUAServer@DESKTOP-UFR9O06	NS4|String||var|CODESYS Control Win V3 x64.Application.GVL.Q1	GlobalVars/GVL/Q1	Q1	false	Boolean	23:01:06.015	23:01:06.015	Good

Figure 6.14: UaExpert's data access view.

6.5 Summary

This chapter has presented all relevant information regarding the tests undertaken in the development of the toolchain outlined in chapter 5.

It begins by highlighting the overall objectives of the tests performed. These objects are: ensuring that the grafcet functions as expected, i.e. if the semantics line up with what is observer on GRAFCET-Studio; ensuring that the translation rules are done according to the rules outlined; ensuring that the internal program variables and the symbols from the symbol configuration from CODESYS are all present and correct.

Following this, a section dedicated to the tests performed on the nine grafcets is shown. It analyses all grafcet tests by first addressing the particular objectives of said test, followed by the description of the grafcet and a figure illustrating the grafcet tested.

The section after is reserved for the tests regarding OPC-UA, where only one test, with the a simple grafcet, is shown for brevity's sake, as only one is needed to fully understand the process.

All tests are confirmed to have been successful given the latest version of the toolchain.

With this, all the main chapters of this work have been exhausted, with the following one presenting the conclusions taken from this endeavor and possible future changes to improve it.

Chapter 7

Conclusion

This chapter presents the conclusion of all the developments of the previous chapters of this work, barring the appendix.

It begins by demonstrating the overall understanding of the problem and the underlying processes which lead to the creation of this work.

Following this, a summary of the relevant knowledge accrued during this work's execution. Describing information regarding the toolchain, the meta model and even referencing the tests performed.

Next, all the potential contributions that this work may have are compiled, considering varied ares of work and technologies.

Finally, the last subsection not only provides multiple possible improvements to the work done, as it also displays some grievances regarding it and the development process.

7.1 Understanding of the Problem

The work presented in this document addresses a critical challenge in industrial automation: the gap between high-level, formal control specifications and their executable implementation on PLCs. GRAFCET is one of multiple standards adopted for modeling sequential control logic, yet PLCs do not directly utilize them, operating on IEC 61131-3 languages.

The absence of automated, semantically faithful translation methods forces engineers to manually convert GRAFCET diagrams into PLC code. This process is error-prone, time-consuming, and compromises the consistency, reliability, and maintainability of the final system.

Moreover, modern trends such as digital twins, modular automation, and Industrial IoT demand tighter integration between specification models and runtime deployments.

Therefore, the core problem lies in the lack of an integrated, automated toolchain capable of transforming GRAFCET specifications, particularly those created in the software denominated GRAFCET-Studio, into standardized, executable IEC 61131-3 code while preserving semantic equivalence and enabling external monitoring.

Taking this into consideration, the toolchain described in this document was created, being capable of processing GRAFCET-Studio grafcets into IEC 61131-3 code and enabling variable-sharing OPC-UA capabilities, utilizing a script to import the necessary information into the CODESYS development environment for PLC execution. Additionally, the toolchain makes use of a meta model structure to ensure grafcet correctness.

7.2 Results and Findings

The work developed and described in this document has yielded a final toolchain and multiple other findings. First, some preliminary considerations, taken from the findings:

- **GRAFCET-Studio Project Structure** - The XML save format was decoded, revealing that logical connections between components must be inferred from graphical coordinates, necessitating a coordinate-based parsing algorithm.
- **PLCOpenXML and IEC 61131-3 Integration** - PLCOpenXML was identified as a vendor-neutral bridge to CODESYS. Structured Text (ST) was chosen as the target language for its expressiveness and suitability for implementing deterministic GRAFCET semantics.
- **Review of Related Work** - The state of the art confirms the need for tools that transform GRAFCET-Studio models while preserving hierarchy and deterministic behavior.

A GRAFCET interpretation algorithm was created, presenting the logical flow for the IEC 61131-3 codes and, additionally, it addresses the grafcet hierarchy problem and other necessary structures for the correct execution of the translated grafcets.

In order to ensure correctness in the created grafcets, the meta model developed for the toolchain was introduced together with the objectives that guided its design. It was defined to accurately represent the elements specified in IEC 60848, providing a structured representation that supports the validation of internal data models while ensuring consistency across the entire toolchain.

Beyond acting as a reference for the verification of imported GRAFCET models, the meta model establishes a well-defined framework for implementing the algorithms responsible for manipulating GRAFCET elements. It also promotes interoperability with future tools by providing a common representation.

The structure of the meta model was presented through a detailed description of its class hierarchy and the relationships between classes. In addition, the representation of GRAFCET elements was described. Collectively, these definitions form the foundation on which the toolchain operates.

As for the implementation of the proposed architecture, the toolchain's first component, the GRAFCET-Studio Reader, parses the save files produced by GRAFCET-Studio, processes them, establishing the inter-component connections , with these being processed by the subsequent stage of the toolchain.

The Structured Text Translator performs this subsequent processing, beginning by parsing the data according to the developed meta model. It further processes the relationships between components, generating partial and connected grafcets, associating them with the corresponding components, and carrying out a meta model analysis to verify the correctness of the internal representation. Afterward, the translation and symbol-processing algorithms are executed, with the resulting information being stored in the JSON format. This output is then combined with a template file to generate the final project files.

The final stage of the toolchain consists of the CODESYS Import Script, which uses the CODESYS API to execute the previously described components sequentially, producing the final output files before automatically importing them into an empty project. The script is executed from within CODESYS to ensure access to the API and the correct resolution of import paths.

Finally, several types of tests performed, with their objectives and the adequate graphical illustrations are presented. These objectives include verifying that the grafcet behaves as expected, ensuring that its semantics match the behavior observed in GRAFCET-Studio, confirming that the implemented translation follows the rules outlined, and validating that both the internal program variables and the symbols defined in the CODESYS symbol configuration are correctly generated.

Apart from testing the grafcets, the OPC-UA integration with CODESYS was also analyzed, with only one test being represented as it is sufficient to illustrate the complete validation procedure. Furthermore, figures are provided to illustrate the symbol structure associated with the test, showing both the CODESYS symbol configuration and the corresponding UaExpert address space.

7.3 Potential Contributions

The contributions which this work is expected to fulfill are as follows:

- **Automated, semantic-preserving code generation** - Providing a reliable mechanism for automatically transforming formal GRAFCET specifications into executable PLC implementations while preserving the original semantics, thereby reducing development effort and minimizing the likelihood of implementation errors.
- **Advancement of model-driven engineering in automation** - Strengthening the adoption of Model-Driven Software Engineering (MDSE) principles in industrial automation by implementing a model-based transformation pipeline that promotes higher levels of abstraction, model reuse, maintainability, and a systematic development methodology.
- **Enhanced design-to-runtime integration** - Establishing a seamless workflow between GRAFCET-Studio, CODESYS, and runtime monitoring systems through the use of PLCOpenXML and OPC-UA, facilitating interoperability across the engineering lifecycle and supporting digital twin applications.

- **Foundation for future extensions** - Establishing a flexible and extensible foundation for future enhancements, including the incorporation of new language features, integration with additional engineering tools, and the adoption of advanced capabilities such as formal verification.
- **Support for education and training** - Offering a pedagogical framework that connects formal GRAFCET models with executable PLC code, enabling students and practitioners to better understand the relationship between formal specifications, model-driven development, and industrial automation.

7.4 Future Improvements

Irrespective of the results obtained and of the quality of the toolchain, there is always the possibility to integrate future improvements upon the tools created.

The overall code developed for the toolchain has proven to be satisfactory for the purposes of the work developed. It can still be greatly optimized by removing redundant checks and reducing the iterative loops used for inspecting elements in the internal data models. Additionally, the data model used in both tools can still be subjected to improvements, although it functions correctly for this work, the fact that all the variables are concentrated into a singular struct may prove inefficient. These may manifest as difficulty in wielding the model when possibly making changes to the existing code or even when attempting to develop another tool which requires intercommunication with the data model or, in the case that the tools are utilized in a grander scope that what is covered in this work, they may prove inefficient, memory-wise, when handling larger models or a plethora of automated tests.

In addition to this, the codebase could be further improved in other aspects, such as the ease of expansion, which could be remedied by the usage of more modular functions and libraries. Tool integration, as referenced before, is also important to mention as, currently, the outputs from both tools can prove bothersome to handle, especially when creating another tool to manage or transform them.

Besides the tests presented in chapter 6, additional tests with different circumstances could have been implemented to analyze other scenarios and real-world applications, further ensuring the toolchain's viability. Additionally, in the interest of making this previous endeavor more feasible, an automatic testing method, requiring the creation of other tools, could prove useful, as manually probing for errors and inconsistencies, especially on more complex grafcets, can lead to inadequate detection.

Moreover, some of the coding conventions utilized for the tools have multiple points which can be further improved, such as:

- The notation convention for variables, functions and other components is sometimes implemented inconsistently.
- How the code is sectioned, be it the main program or the libraries created.

- Inconsistent comments and code block structures.
- Similar types of functions having different types of approaches and return values, e.g. a function that finds a given element based on its ID and another on its name, the first returns the element object whilst the second returns its ID.

Furthermore, the error checking is considerably lacking in both tools, as they heavily rely on the source material being correct, given the non-existence of some edge case checks and the lack of simple input checking for most functions:Debugging is also problematic, as the current debug prints are relatively disorganized, lacking structure and being at times hard to interpret.

Particularly, the Structured Text Translator tool contains some code block repetition, where instead generalist functions should be utilized. Additionally, the JSON model created proves difficult to work with when more complex trees are considered as editors lack methods of demonstrating the structure with code completion, akin to the IntelliSense of Visual Studio Code, a shortcoming of the library used which was not considered. This problem could have been addressed via the use of nested structs with the according entries equivalent to the ones present in the JSON model and, at the end of the assembly phase, transform the struct model into a JSON model to be used right away on the file creating phase. In addition to this, although the internal model that utilizes a singular struct has proven itself to be simpler, at least for this second tool, it should have reflected the meta model developed more closely, if not a one to one representation. For this, Object Oriented Programming (OOP) could have been helpful, especially when considering the possibility of incorporating component specific methods into the objects similarly to how some functions were employed; simple structs associated to one another would probably have worked just as well.

A possible addition to the toolchain would be the usage of a Graphical User Interface (GUI). Unlike other concepts mentioned in this section, its usefulness would not lie in its ability to improve code performance or integration, instead it would make the toolchain more user friendly, implementing file navigation for both selecting the input files and saving the output files, which could be in different directories from the tool. Additionally, it could provide with a more visually appealing debug menu and error notifications, outlining the possible culprits. Providing this GUI program with the ability to integrate with CODESYS to facilitate the importing process could prove challenging but also be useful.

In summary, this work has provided a practical, standard-compliant solution to a persistent integration problem. By automating the GRAFCET-to-IEC 61131-3 transformation, it aims to enhance productivity, reliability, and connectivity in modern automation projects.

References

[1] Alexander Larsson. *Dia*. http://dia-installer.de/. Accessed: Jul. 2026.

[2] Oriol Bayó-Puxan et al. "A GRAFCET-compiler Methodology for C-programmed Microcontrollers". In: *Assembly Automation* 28.1 (Feb. 2008), pp. 55–60. ISSN: 0144-5154. DOI: 10.1108/01445150810849019. (Visited on 10/19/2025).

[3] Chih-Hong Cheng, Chung-Hao Huang, Harald Ruess, Stefan Stattelmann. *G4LTL-ST*. https://arxiv.org/abs/1405.2409. Accessed: Jun. 2026.

[4] CODESYS GmbH. *CODESYS Development System V3*. https://www.codesys.com. Accessed: Jan. 2026.

[5] René David and Hassane Alla. *Petri nets and grafcet : tools for modelling discrete event systems / René David, Hassane Alla*. eng. New York: Prentice-Hall, 1992. ISBN: 0-13-327537-X.

[6] Eclipse Foundation AISBL. *Eclipse Modeling Framework*. https://eclipse.dev/emf/. Accessed: Jun. 2026.

[7] Emmanuel Geveaux Software. *Grafcet Designer*. https://www.ni.com/en/support/downloads/tools-network/download.grafcet-designer.html#372321. Accessed: Jun. 2026.

[8] Famic Technologies Inc. *Automation Studio*. https://www.famictech.com/en/Products/Automation-Studio/Professional-Edition. Accessed: Jun. 2026.

[9] Festo Didactic. *FluidSIM*. https://www.festo.com/pt/pt/e/formacao-tecnica/aprendizagem-digital/simulacao-e-modelacao-virtuais/fluidsim-id_1663056/. Accessed: Jun. 2026.

[10] Florentin Delaine. *grafcet2plc*. https://github.com/fdelaine/grafcet2plc. Accessed: Jun. 2026.

[11] Ingenieurbüro Dr .- Ing. Schoop GmbH. *WinErs*. https://www.schoop.de/en/didactic/winers-laboratory-versions/grafcet-laboratory-version. Accessed: Jun. 2026.

[12] International Electrotechnical Commission. *IEC 60848:2013 – Specification language GRAFCET for sequential function charts*. Geneva, Switzerland: IEC, 2013.

[13] International Electrotechnical Commission. *IEC 61131-3:2013 – Programmable controllers – Part 3: Programming languages*. Geneva, Switzerland: IEC, 2013.

[14] Robert Julius et al. "A Meta-Model Based Environment for GRAFCET Specifications". In: *2019 IEEE International Systems Conference (SysCon)*. 2019 IEEE International Systems Conference (SysCon). Orlando, FL, USA: IEEE, Apr. 2019, pp. 1–7. ISBN: 978-1-5386-8396-5. DOI: 10.1109/SYSCON.2019.8836959. URL: https://ieeexplore.ieee.org/document/8836959/ (visited on 02/03/2026).

[15] Robert Julius et al. “A Model-Driven Approach for Transforming GRAFCET Specification into PLC Code Including Hierarchical Structures”. In: *IFAC-PapersOnLine* 52.13 (2019), pp. 1767–1772. ISSN: 24058963. DOI: 10.1016/j.ifacol.2019.11.457. URL: https://linkinghub.elsevier.com/retrieve/pii/S2405896319314387 (visited on 10/21/2025).

[16] Robert Julius et al. “Transformation of GRAFCET to PLC Code Including Hierarchical Structures”. In: *Control Engineering Practice* 64 (July 2017), pp. 173–194. ISSN: 09670661. DOI: 10.1016/j.conengprac.2017.03.012. URL: https://linkinghub.elsevier.com/retrieve/pii/S0967066117300709 (visited on 10/12/2025).

[17] Luis Garcia, Stefan Mitsch, Andre Platzer. *HyPLC*. https://arxiv.org/abs/1902.05205. Accessed: Jun. 2026.

[18] MHJ-Software GmbH & Co. KG. *GRAFCET-Studio*. https://www.mhj-tools.com/?page=grafcet-studio. Accessed: Jan. 2026.

[19] Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros Araya, Oluwatosin Ogundare, Mohammad Abdullah Al Faruque. *LLM4PLC*. https://arxiv.org/abs/2401.05443, https://sites.google.com/uci.edu/llm4plc/home. Accessed: Jun. 2026.

[20] R. Mroß et al. “Model Checking Generated Control Code for Consistency with Its Specification in IEC 60848 GRAFCET”. In: *IFAC-PapersOnLine* 58.1 (2024), pp. 264–269. ISSN: 24058963. DOI: 10.1016/j.ifacol.2024.07.045. URL: https://linkinghub.elsevier.com/retrieve/pii/S240589632400123X (visited on 06/08/2026).

[21] Robin Mroß et al. “Unambiguous Interpretation of IEC 60848 GRAFCET Based on a Literature Review”. In: *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*. Sept. 12, 2023, pp. 1–8. DOI: 10.1109/ETFA54631.2023.10275504. arXiv: 2307.11556 [eess]. URL: http://arxiv.org/abs/2307.11556 (visited on 10/19/2025).

[22] Robin Mross et al. “Transformation of GRAFCET Into GAL for Verification Purposes Based on a Detailed Meta-Model”. In: *IEEE Access* 10 (2022), pp. 125652–125665. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3225736. URL: https://ieeexplore.ieee.org/document/9966565/ (visited on 10/12/2025).

[23] National Instruments. *LabVIEW*. https://www.ni.com/en/shop/labview. Accessed: Jun. 2026.

[24] Obeo. *Acceleo*. https://eclipse.dev/acceleo/. Accessed: Jun. 2026.

[25] Ofek Glick, Vladimir Tchuiev, Marah Ghoummaid, Michal Moshkovitz, Dotan Di-Castro. *LLM4SFC*. https://arxiv.org/abs/2512.06787. Accessed: Jun. 2026.

[26] Omegon Teachware. *OFT2*. https://www.omesim.com/softwareoft2/oft2-grafcet-version/. Accessed: Jun. 2026.

[27] OPC Unified Architecture. *(Open Platform Communications Unified Architecture*. OPC Foundation, 2026. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/.

[28] Xoán C. Pardo Martínez and Ramón Ferreiro García. “Initial Design of a Virtual Machine to Play Grafcet Models”. In: *IFAC Proceedings Volumes* 33.17 (July 2000), pp. 41–46. ISSN: 14746670. DOI: 10.1016/S1474-6670(17)39372-2. URL: https://linkinghub.elsevier.com/retrieve/pii/S1474667017393722 (visited on 02/03/2026).

[29] PLCopen. *PLCopen – The worldwide association for IEC 61131-3*. https://www.plcopen.org. Accessed: Jan. 2026.

[30] PLCopen. *PLCopen XML – Technical Specification*. https://www.plcopen.org/standards/xml-echange/. Accessed: Jan. 2026.

[31] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. "A Formal Semantics for Grafcet Specifications". In: *2011 IEEE International Conference on Automation Science and Engineering*. 2011 IEEE International Conference on Automation Science and Engineering (CASE 2011). Trieste, Italy: IEEE, Aug. 2011, pp. 488–494. ISBN: 978-1-4577-1730-7. DOI: 10.1109/CASE.2011.6042457. URL: http://ieeexplore.ieee.org/document/6042457/ (visited on 10/19/2025).

[32] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. "Translating Grafcet Specifications into Mealy Machines for Conformance Test Purposes". In: *Control Engineering Practice* 19.9 (Sept. 2011), pp. 947–957. ISSN: 09670661. DOI: 10.1016/j.conengprac.2010.10.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0967066110002108 (visited on 11/21/2025).

[33] Max Schürenberg. "Transformation of GRAFCET-Based Control Specifications Into an IEC 61131-3 Implementation". In: (2015).

[34] Stéphane Di Meglio. *SFCEDIT*. https://web.archive.org/web/20190910022119/http://stephane.dimeglio.free.fr/sfcedit-en.html. Accessed: Jul. 2026.

[35] Kleanthis Thramboulidis and Georg Frey. "Towards a Model-Driven IEC 61131-Based Development Process in Industrial Automation". In: *Journal of Software Engineering and Applications* 04.04 (2011), pp. 217–226. ISSN: 1945-3116, 1945-3124. DOI: 10.4236/jsea.2011.44024. URL: https://www.scirp.org/journal/paperinformation?paperid=4753 (visited on 06/08/2026).

[36] Unified Automation GmbH. *UaExpert*. https://www.unified-automation.com/products/development-tools/uaexpert.html. Accessed: Jul. 2026.

[37] Birgit Vogel-Heuser et al. "Model-Driven Engineering of Manufacturing Automation Software Projects – A SysML-based Approach". In: *Mechatronics* 24.7 (Oct. 2014), pp. 883–897. ISSN: 09574158. DOI: 10.1016/j.mechatronics.2014.05.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0957415814000853 (visited on 06/08/2026).

[38] Nils Weidmann, Suganya Kannan, and Anthony Anjorin. *Tolerance in Model-Driven Engineering: A Systematic Literature Review with Model-Driven Tool Support*. June 2, 2021. DOI: 10.48550/arXiv.2106.01063. arXiv: 2106.01063 [cs.SE]. URL: http://arxiv.org/abs/2106.01063 (visited on 06/08/2026). Pre-published.

Appendix A

Toolchain User Manual

This chapter addresses the user manual regarding the toolchain created for this work. It begins by describing the installation process for the toolchain's files in section A.1. Following this is the description of the toolchain usage in section A.2.

A.1 Toolchain Installation

The installation of the toolchain begins in its dedicated GitHub repository, selecting the "Releases" section as indicated by figure A.1.

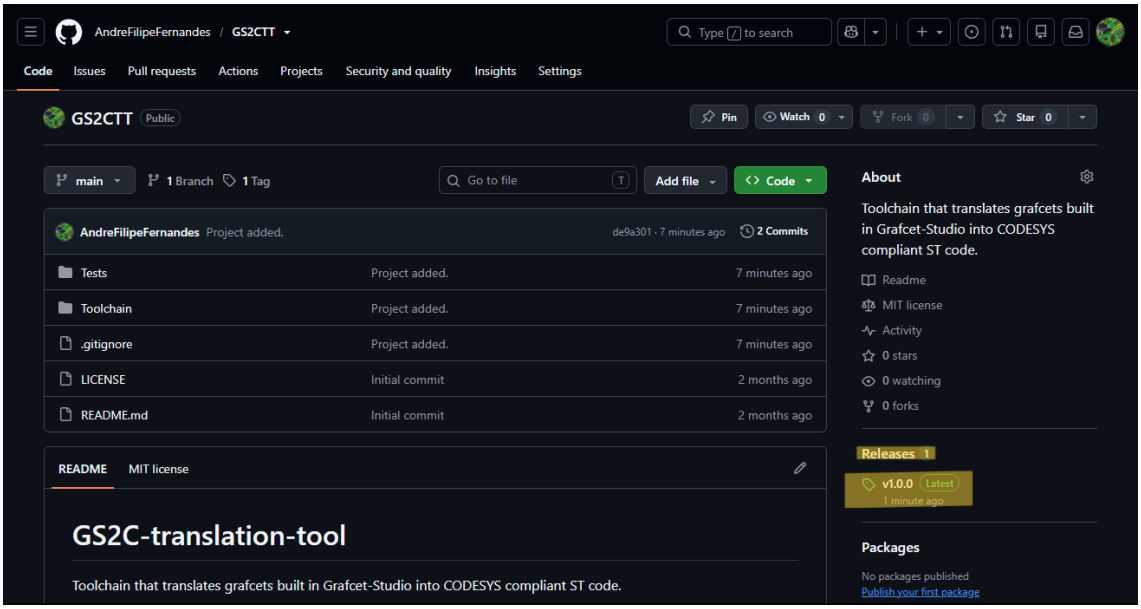


Figure A.1: GitHub page for the toolchain with the "releases" section highlighted.

Afterwards, within the releases page - figure A.2 - download the source code for the latest version of the toolchain by clicking "Source Code (zip)" if using a Windows Operating System (OS), otherwise, for a Linux OS select "Source code (tar.gz)" instead. Note that the toolchain has

been designed with Windows in consideration, meaning that using it with other operating systems may result in problems.

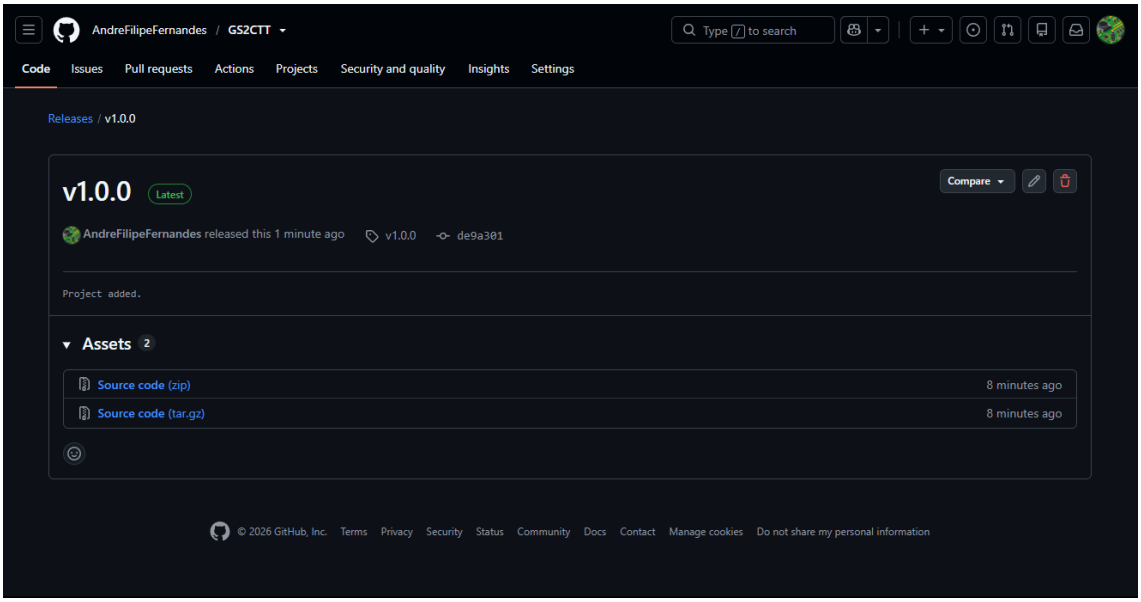


Figure A.2: Releases page.

The final step of the installation is to simply extract the downloaded compressed file into any desired directory. The example in figure A.3 demonstrates the final product of this extraction for the toolchain version "v1.0.0". Avoid using the toolchain from within privilege-protected directories as its behavior was not tested under such circumstances.

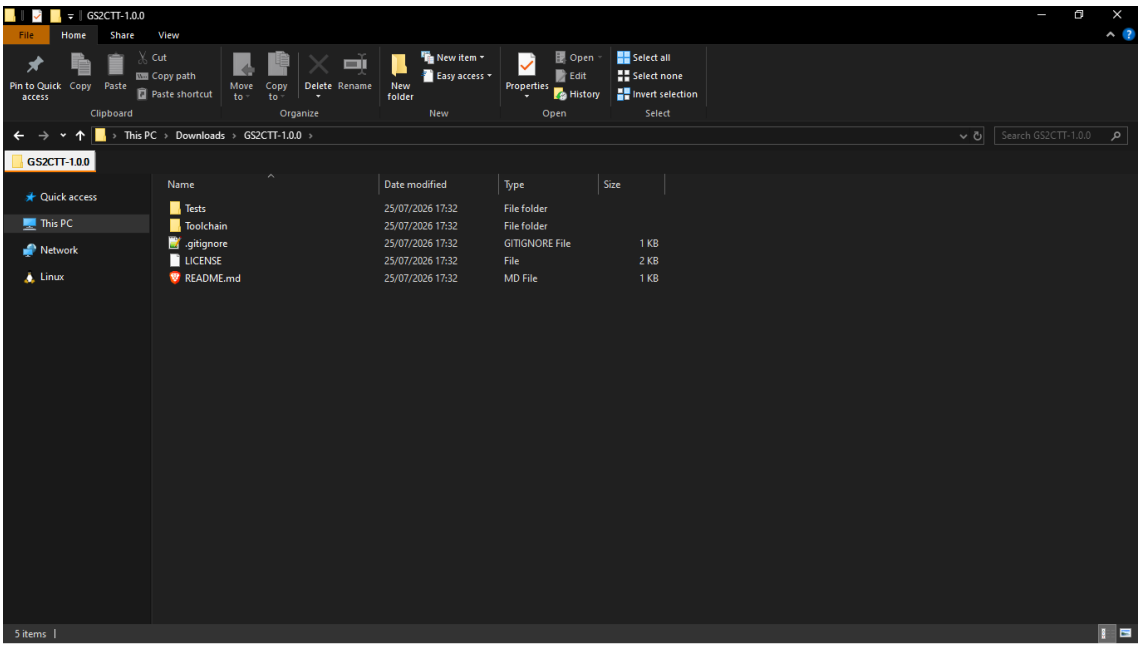


Figure A.3: Example directory of successful toolchain zip extraction.

A.2 Toolchain Usage

After the toolchain's installation has been completed, the first step of the usage process is to ensure that the toolchain is ready to run. This begins by checking whether the "main.exe" executables exist in their appropriate "src" directories for both of the tools and checking whether the "shared" directory exists. If any of these checks are proven to be false, then the compilation batch script present in figure A.4, "toolchain_compile.bat", must be ran, by either double-clicking it or running via console call, therefore creating the executables and the directory. A new installation will always need to run said script once before any other step is taken.

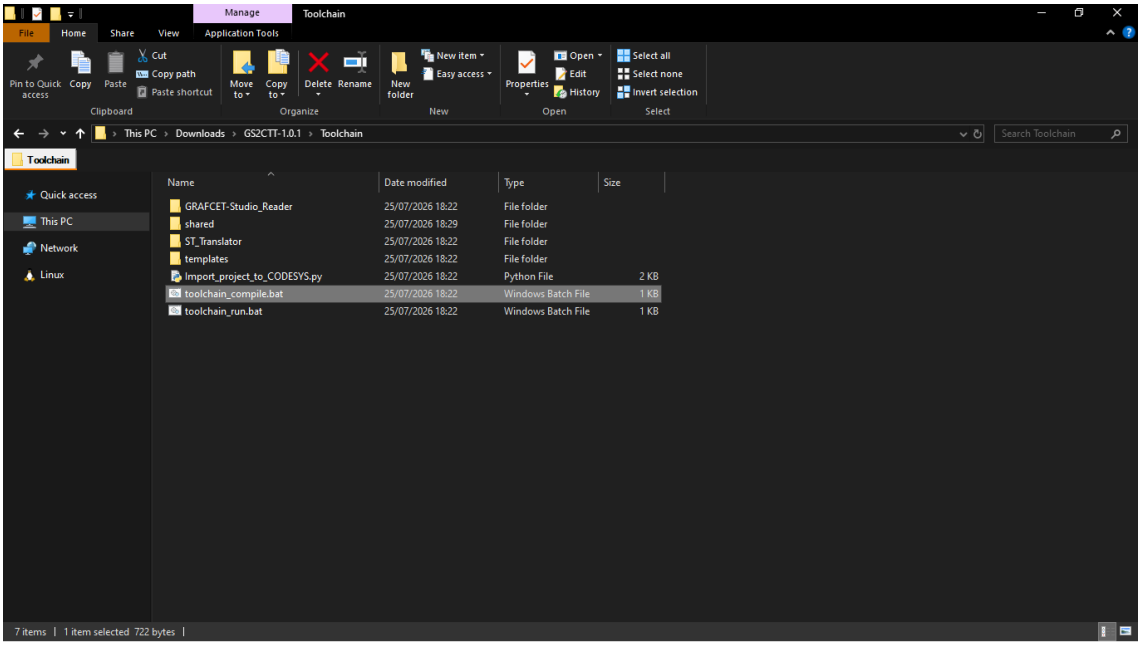


Figure A.4: Toolchain compilation script "toolchain_compile.bat".

Secondly, in order to use the toolchain a grafcet chart must be provided. As such, the next step is to create a grafcet utilizing GRAFCET-Studio and saving it to the "shared" directory from the toolchain as figure A.5 shows.

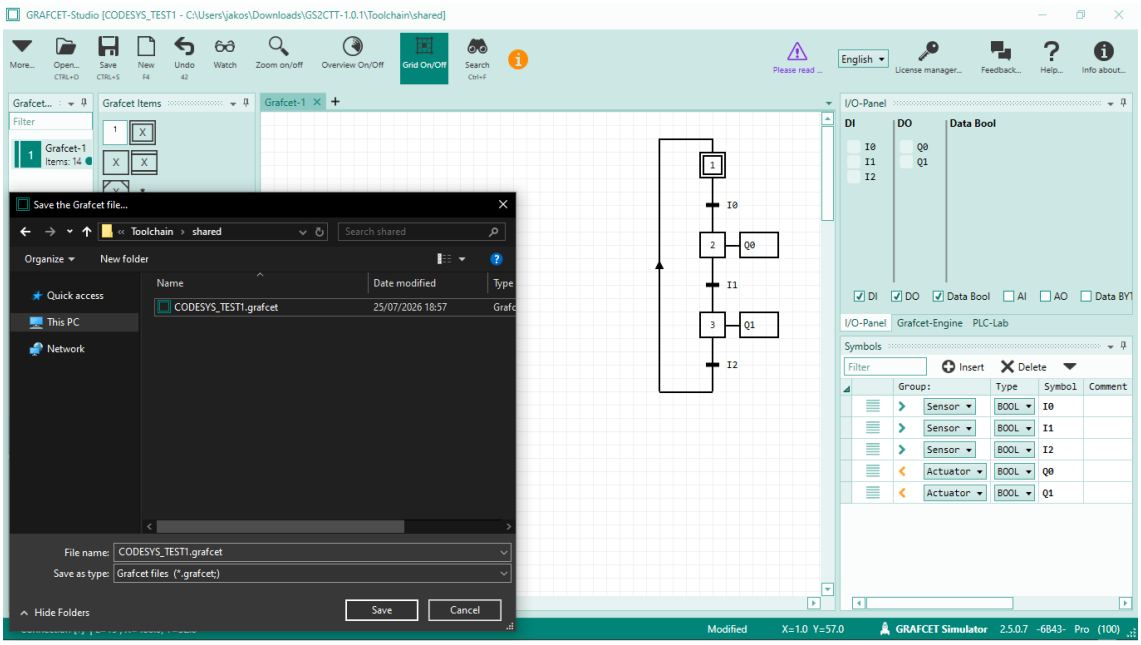


Figure A.5: Creation and saving of grafcet chart in "shared" directory.

Finally, the last phase of the toolchain usage begins. After opening CODESYS, create an empty project with any name and in any location - figure A.6. Note that an empty project is utilized here solely for demonstration purposes as the project files can still be imported into an already existing CODESYS project environment; however, this aspect of the import script was not thoroughly tested, possibly resulting in errors.

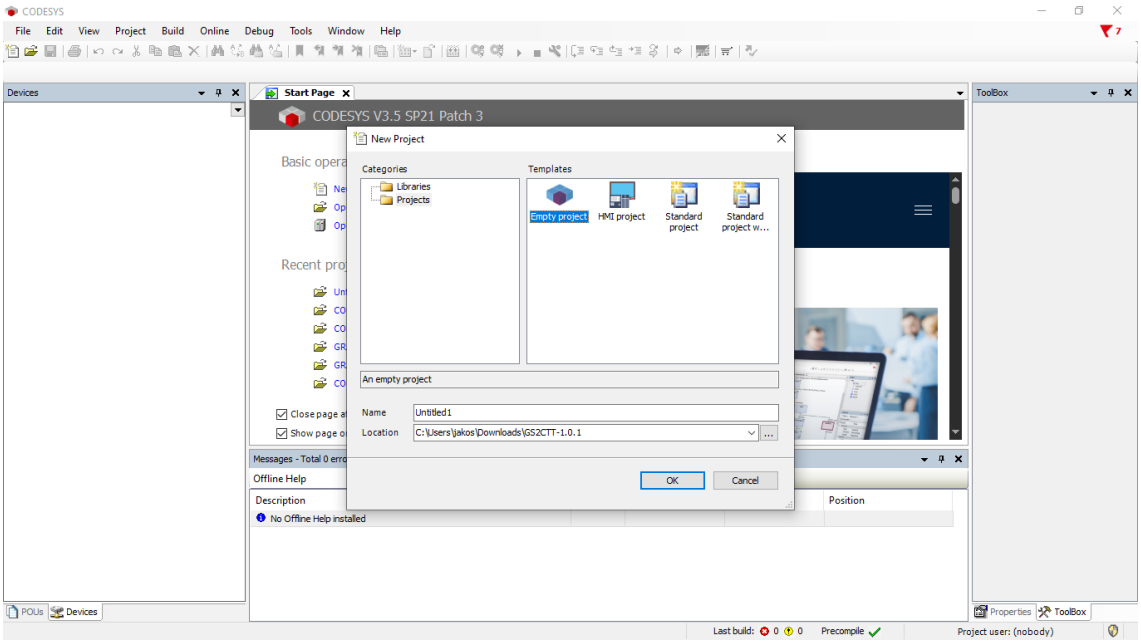


Figure A.6: Creation of empty CODESYS project.

Following this, the CODESYS Import Script, "Import_project_to_CODESYS.py", is ran from

within its editing environment. As per the version of CODESYS utilized for the creation of this toolchain (CODESYS V3.5 SP21 Patch 3), the script-running utility is located in the CODESYS toolbar above, under "Tools" > "Scripting" > "Execute Script File...", as demonstrated in figure A.7.

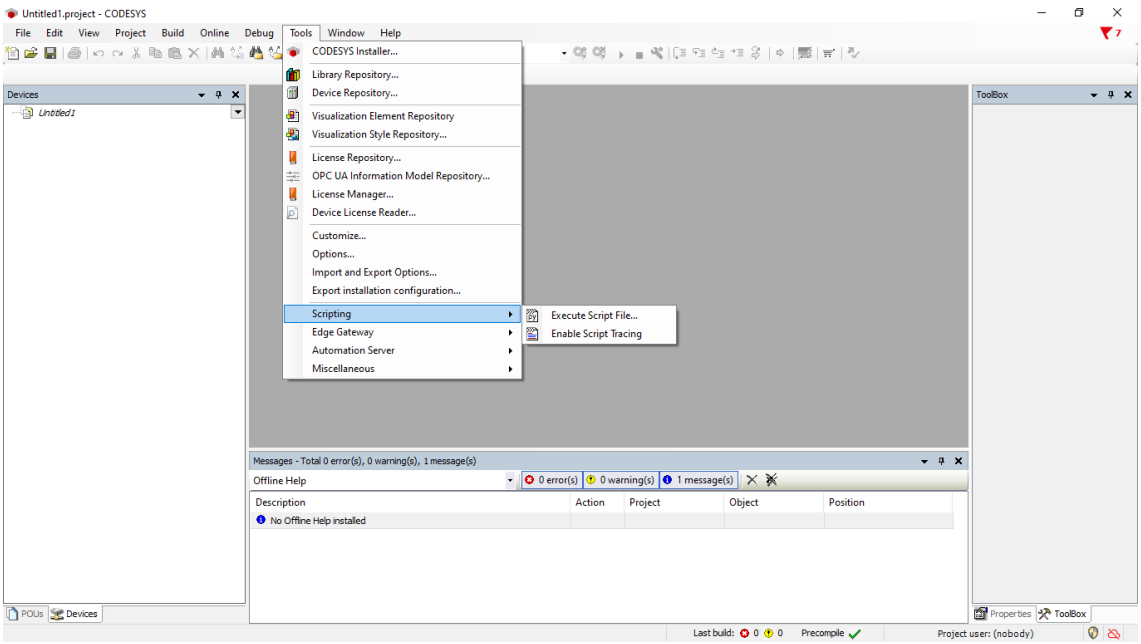


Figure A.7: Execution of the CODESYS Import Script.

This concludes the toolchain workflow. The resulting CODESYS project for this example is shown in figure A.8.

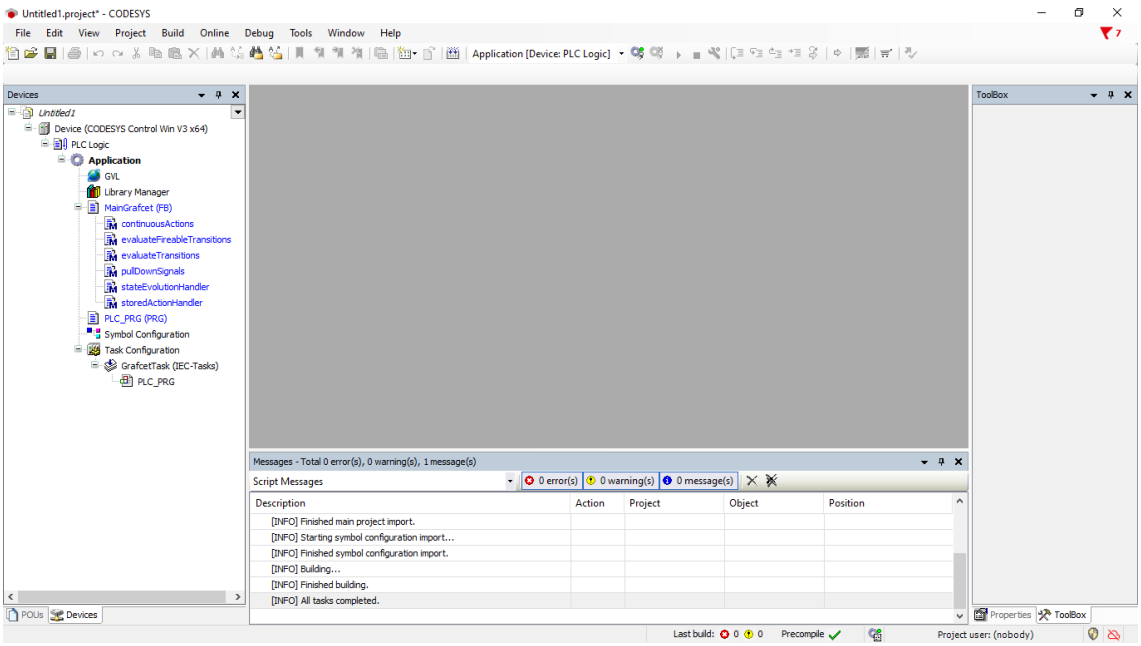


Figure A.8: Resulting CODESYS project after script has finished.

Appendix B

Toolchain File Structure Listings

This chapter addresses the listings referenced in chapter 5 for the output files of the Structured Text Translator tool. Note that in this chapter the use of "(...)" indicates that some text was omitted or that the remaining content continues on the following listing or is the continuation of the previous listing.

B.1 CODESYSProject.xml File

This section shows the listings regarding the "CODESYSProject.xml" file.

Listing B.1: "CODESYSProject.xml" file's task structure

```
<task name="{{␣task.name␣}}␣priority="1">
␣␣<pouInstance␣name="{{ plcprg.name }} typeName="program"></
    pouInstance>
  <addData>
    <data name="http://www.3s-software.com/plcopenxml/
        tasksettings" handleUnknown="implementation">
      <TaskSettings KindOfTask="Freewheeling" Interval="t#20ms"
          IntervalUnit="ms" WithinSPSTimeSlicing="true">
        <Watchdog Enabled="false" TimeUnit="ms" Sensitivity="1"
            />
      </TaskSettings>
    </data>
    <data name="http://www.3s-software.com/plcopenxml/objectid"
        handleUnknown="discard">
      <ObjectId>{{ task.UUID }}</ObjectId>
    </data>
  </addData>
</task>
```

Listing B.2: "CODESYSProject.xml" file's global variable list structure

```
<globalVars name="{{ gvl.name }}>
## for gvlVarFB in gvl.variable_list.function_blocks
  {{ gvlVarFB }}
## endfor
## for gvlVarOther in gvl.variable_list.others
  {{ gvlVarOther }}
## endfor
  <addData>
    <data name="http://www.3s-software.com/plcopenxml/objectid"
      handleUnknown="discard">
      <ObjectId>{{ gvl.UUID }}</ObjectId>
    </data>
  </addData>
</globalVars>
```

Listing B.3: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 1

```
<data name="http://www.3s-software.com/plcopenxml/pou"
  handleUnknown="implementation">
    <pou name="{{ plcprg.name }}" pouType="program">
      <interface>
        <localVars>
          <variable name="globalStableState">
            <type>
              <BOOL />
            </type>
            <initialValue>
              <simpleValue value="FALSE" />
            </initialValue>
          </variable>
          <variable name="startProgram">
            <type>
              <BOOL />
            </type>
            <initialValue>
              <simpleValue value="TRUE" />
            </initialValue>
          </variable>
        </localVars>
      </interface>
```

```
(...)
```

Listing B.4: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.1

```
(...)
                <body>
                  <ST>
                    <xhtml xmlns="http://www.w3.org/1999/xhtml">
                      // Check triggers
## for trig in plcprg.body.fb_triggerHandler
{{ trig }}
## endfor
## if plcprg.body.fb_triggerHandler | length == 0
// ...
## endif
(...)
```

Listing B.5: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.2

```
(...)
// Check time conditions
## if plcprg.body.fb_timedConditionHandler | length
## for time in plcprg.body.fb_timedConditionHandler
{{ time }}
## endfor
## else
// ...
## endif
(...)
```

Listing B.6: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.3

```
(...)
// Initial stored action handling owing to initial steps
IF startProgram = TRUE THEN
  startProgram := FALSE;
## if plcprg.body.fb_stepTriggerHandler | length
## for trig in plcprg.body.fb_stepTriggerHandler
  {{ trig }}
## endfor
## endif
## for sAction1 in plcprg.body.fb_storedActionHandler
  {{ sAction1 }}
```

```
## endfor
END_IF;
(...)
```

Listing B.7: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.4

```
(...)
// Check if transition condition is fireable, saves result to
   variable
## for evalT in plcprg.body.fb_evaluateTransitions
{{ evalT }}
## endfor
(...)
```

Listing B.8: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.5

```
(...)
// Loop while stable state for ALL grafcets is not reached
globalStableState := FALSE;
WHILE globalStableState = FALSE DO
  globalStableState := TRUE;

  // Check transition state according to previously established
     condition and previous step state
## for evalFT in plcprg.body.fb_evaluateFireableTransitions
  {{ evalFT }}
## endfor

  // State evolution handling
## for sEvolH in plcprg.body.fb_stateEvolutionHandler
  {{ sEvolH }}
## endfor

  // Activate macros/enclosing expansions
## if plcprg.body.fb_activateExpansions | length == 0
  // ...
## else
## for actExp in plcprg.body.fb_activateExpansions
  {{ actExp }}
## endfor
## endif
```

```
  // Check step triggers
## if plcprg.body.fb_stepTriggerHandler | length
## for trig in plcprg.body.fb_stepTriggerHandler
  {{ trig }}
## endfor
## endif

  // Check stored actions
## for sAction in plcprg.body.fb_storedActionHandler
  {{ sAction }}
## endfor
END_WHILE;
(...)
```

Listing B.9: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 2.6

```
(...)
// Turn forcedFlag and all continuous action states to FALSE
## for pDSignal in plcprg.body.fb_pullDownSignals
{{ pDSignal }}
## endfor

// Check continuous actions
## for cAction in plcprg.body.fb_continuousActions
{{ cAction }}
## endfor
</xhtml>
                  </ST>
                </body>
(...)
```

Listing B.10: "CODESYSProject.xml" file's "PLC_PRG" structure - Part 3

```
(...)
                <addData>
                  <data name="http://www.3s-software.com/
                      plcopenxml/objectid" handleUnknown="discard
                      ">
                    <ObjectId>{{ plcprg.UUID }}</ObjectId>
                  </data>
                </addData>
              </pou>
```

```
            </data>
```

Listing B.11: "CODESYSProject.xml" file's POU structure - Part 1

```
## for pou in pou_list
            <data name="http://www.3s-software.com/plcopenxml/
               pou" handleUnknown="implementation">
              <pou name="{{ pou.name }}" pouType="functionBlock"
                 >
                <interface>
                  <localVars>
                    {% if pou.misc_variable_list | length %}
                    {% for var in pou.misc_variable_list -%}
                    {{ var }}
                    {%- endfor -%}
                    {% endif %}
                    {% if pou.transition_variable_list | length
                       %}
                    {% for var in pou.transition_variable_list
                       -%}
                    {{ var }}
                    {%- endfor -%}
                    {% endif %}
                  </localVars>
                </interface>
(...)
```

Listing B.12: "CODESYSProject.xml" file's POU structure - Part 2

```
(...)
                <addData>
                  {%- for method in pou.method_list %}
                  <data name="http://www.3s-software.com/
                     plcopenxml/method" handleUnknown="
                     implementation">
                    <Method name="{{ method.name }}" ObjectId="
                       {{ method.UUID }}">
                     <interface>
                     {%- if method.return_type | length -%}
                     <returnType>
                       <{{ method.return_type }} />
                     </returnType>
```

```
                        {%- endif -%}
                        {%- if method.variable_list | length -%}
                          <inputVars>
                            {% for var in method.variable_list -%}
                              {{ var }}
                            {%- endfor %}
                          </inputVars>
                        {%- endif -%}
                        </interface>
                        <body>
                          <ST>
                            <xhtml xmlns="http://www.w3.org/1999/
                                xhtml">{% if method.body | length
                                %}{% for line in method.body %}{{
                                line }}{% endfor %}{% endif %}</
                                xhtml>
                          </ST>
                        </body>
                        <addData />
                      </Method>
                    </data>
                    {% endfor -%}
(...)
```

Listing B.13: "CODESYSProject.xml" file's POU structure - Part 3

```
(...)
                    <data name="http://www.3s-software.com/
                        plcopenxml/objectid" handleUnknown="discard
                        ">
                      <ObjectId>{{ pou.UUID }}</ObjectId>
                    </data>
                  </addData>
                </pou>
              </data>
## endfor
```

B.2 SymbolConfiguration.xml File

This section shows the listing regarding the "SymbolConfiguration.xml" file.

Listing B.14: "SymbolConfiguration.xml" file's symbol structure

```
<List2 Name="SelectionSigns">
## if symConfig.entries | length
## for ent in symConfig.entries
          <Single Type="{a7616865-54da-4628-9587-695c8e270109}"
             Method="IArchivable">
            <Single Name="SignName" Type="string">{{ ent.name }}
               </Single>
            <Single Name="SignLibIdIndex" Type="int">-1</Single>
            <List2 Name="SignVariables">
## if ent.variable_list | length
## for var in ent.variable_list
              {{ var }}
## endfor
## endif
            </List2>
            (...)
## endfor
## endif
        </List2>
```

Appendix C

JSON Model

The internal JSON model used in the Structured Text Translator tool to store and structure the information that is later transcribed to the save files using the corresponding templates is described here. The model is divided into two parts.

The first part, depicted in figure C.1, contains the overall project metadata, the tasks, the main POU that makes use of the interpretation algorithm, the global variable list, and the list of supporting POUs.

The metadata related portion of this first part addresses the name of the project ("project_name"), its time of creation ("time_of_creation") and three UUIDs used for other components ("libman_-UUID", "application_UUID", "device_UUID").

As for the task information ("task") there is only its name ("task" > "name") and its UUID ("task" > "UUID").

Regarding the "PLC_PRG" POU ("plcprg"), it contains a name ("plcprg" > "name"), UUID ("plcprg" > "UUID") and a body ("plcprg" > "body"). The "body" contains all the lines of code with the appropriate method calls and supporting structures, in string format, which address the GRAFCET interpretation algorithm.

The global variable list ("gvl") provided has a name ("gvl" > "name"), UUID ("gvl" > "UUID") and a variable list ("gvl" > "variable_list"). This variable list further contains two sections, one pertaining to the variables that instantiate the function blocks ("gvl" > "variable_list" > "function_blocks"), and the other for every other variable that may have to be written in the GVL ("gvl" > "variable_list" > "others").

Finally, the POU list ("pou_list"), whose graphical representation actually indicates the structure elements for a singular POU. As the name implies, this list contains multiple POUs with their variables represented in figure C.1; containing all the basic elements: a name ("pou_list" > "name") and a UUID ("pou_list" > "UUID").

Furthermore containing a list with all the declarations, written as strings and formatted as XML, of the step variables ("pou_list" > "step_variable_list"), the transition related variables ("pou_list" > "transition_variable_list") and of other miscellaneous variables, mostly triggers and time conditions ("pou_list" > "misc_variable_list").

It additionally has a method list ("pou_list" > "method_list"), a list where all the appropriate methods and their variables are contained. More specifically, this last list contains, per method, a name ("pou_list" > "method_list" > "name"), an UUID ("pou_list" > "method_list" > "UUID"), a body ("pou_list" > "method_list" > "body"), a return type ("pou_list" > "method_list" > "return_type") and a list of variables ("pou_list" > "method_list" > "variable_list"). This body functions much the same as the body of the "PLC_PRG", containing the code which will eventually be executed, as does the list of variables contain the method related variables. Moreover, the return type represents just that, the type of the variable which is returned by the method post execution.

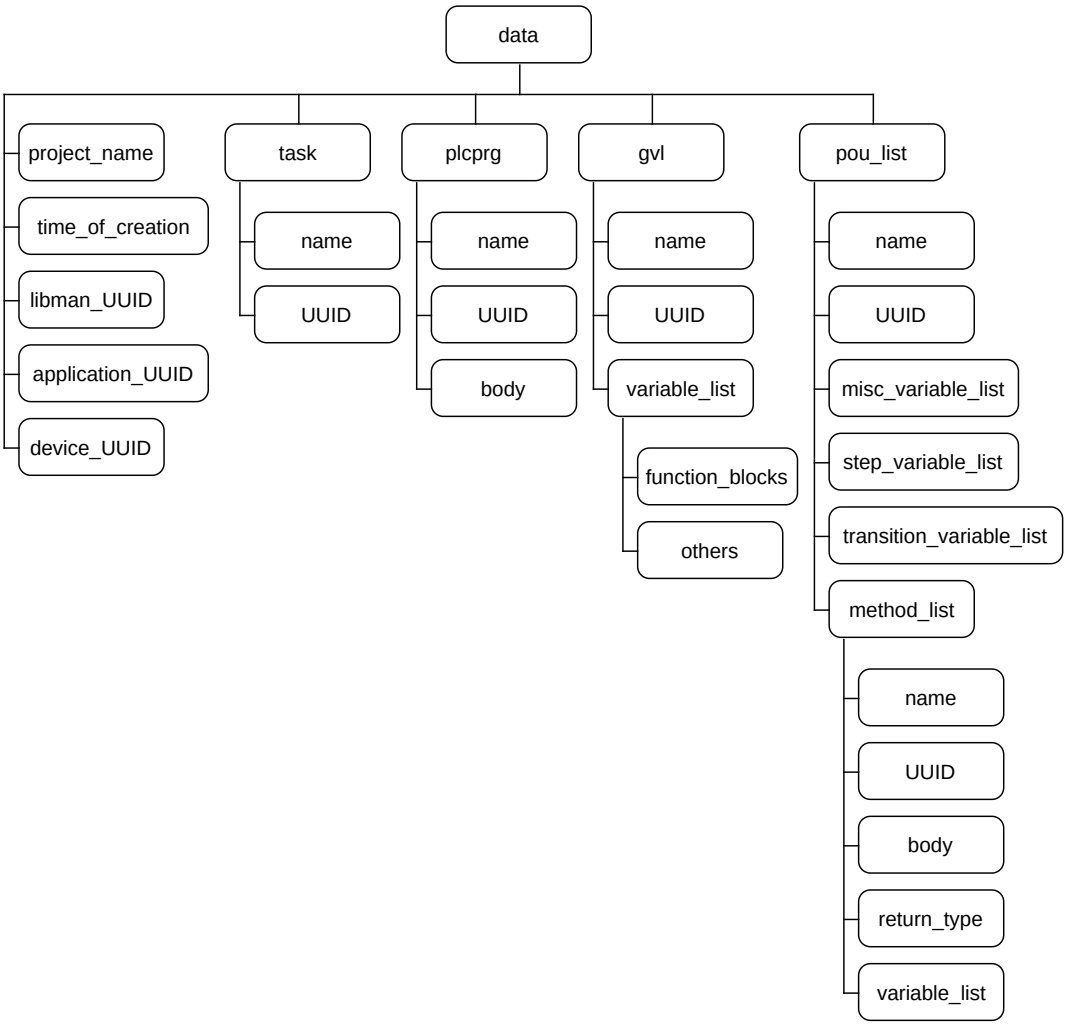


Figure C.1: Diagram depicting first part of JSON structure.

The second part of this model describes the required data for the symbol configuration aspect of the toolchain, containing the necessary symbol entries and assorted UUIDs - see figure C.2.

It provides the necessary symbol information via a symbol configuration element ("symConfig"), which itself contains a list of UUIDs ("symConfig" > "randomUUIDs") used for the miscellaneous CODESYS components of the save file template and a list of all symbol entries ("sym-

Config" > "entries"). The list of entries has the information regarding all the processed symbols, containing the name of a given entry ("symConfig" > "entries" > "name") to which the symbols belong to, e.g. the GVL. Each entry additionally contains a list with all the variables/symbols ("symConfig" > "entries" > "variable_list"), a list with all the variables information in strings formatted as XML.

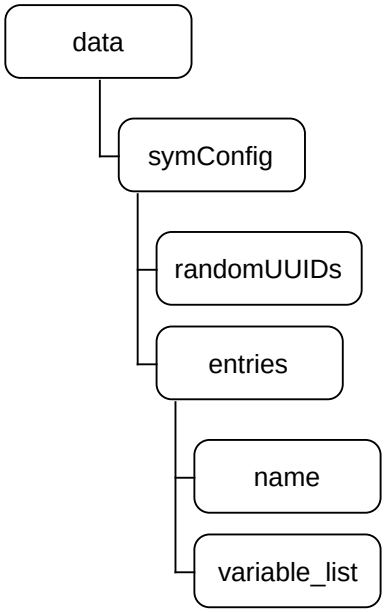


Figure C.2: Diagram depicting second part of JSON structure.

Appendix D

GRAFCET Transformation Rules

This chapter describes the GRAFCET translation rules and the algorithm that dictates the order in which they are to be processed. These were defined in order to aid the translation process of the Structured Text Translator tool.

These are built in accordance with the algorithm developed in the work of Schürenberg et al., described by its figure 5.6 on page 52 and by more in-depth information in its appendix section A.2 [33]. The algorithm is described here, in the following two figures, with the "rule" annotations referring to the rules of component translation developed by Schürenberg - figures D.1 and D.2.

The first part of the translation algorithm illustrated in the following activity diagram - figure D.1 - goes over the condition for which to apply the translation rules to both transitions and steps. For this particular case the translation order is irrelevant, so the order provided in [33] is used.

Starting with the transition translation rules, in which the tool's developed algorithm makes use of function **build_transitions_template()**. This function analyses all transitions belonging to a connected grafcet, acquiring all the next components in the sequence (even in the presence of a parallelization), checking afterwards the transition type. If it is a normal or sink types, it checks whether the previous component is a macro step, a synchronization or any other step type, applying the according rules of translation and saving them onto the JSON model. If on the other hand the transition type is source, rule 6 or 7 is applied without regard for previous sequence components, as there should be none.

Following the transition processing is the step processing, where the **build_steps_template()** function is used for the application of the step translation rules. The function first determines the type of partial grafcet being analyzed, then iterating through all the steps of a given connected grafcet from said partial grafcet. It checks if the type of the step is normal, normal and initial, enclosing, enclosing and initial or macro, deciding on which rule to apply based on their type and the partial grafcet type as well. Moreover, some edge cases exist, one may occur to any step type where an activation link is present - rules 26 and 27 - whilst the other occurs only when a normal step happens to be the entry step for a macro step expansion - rule 28.

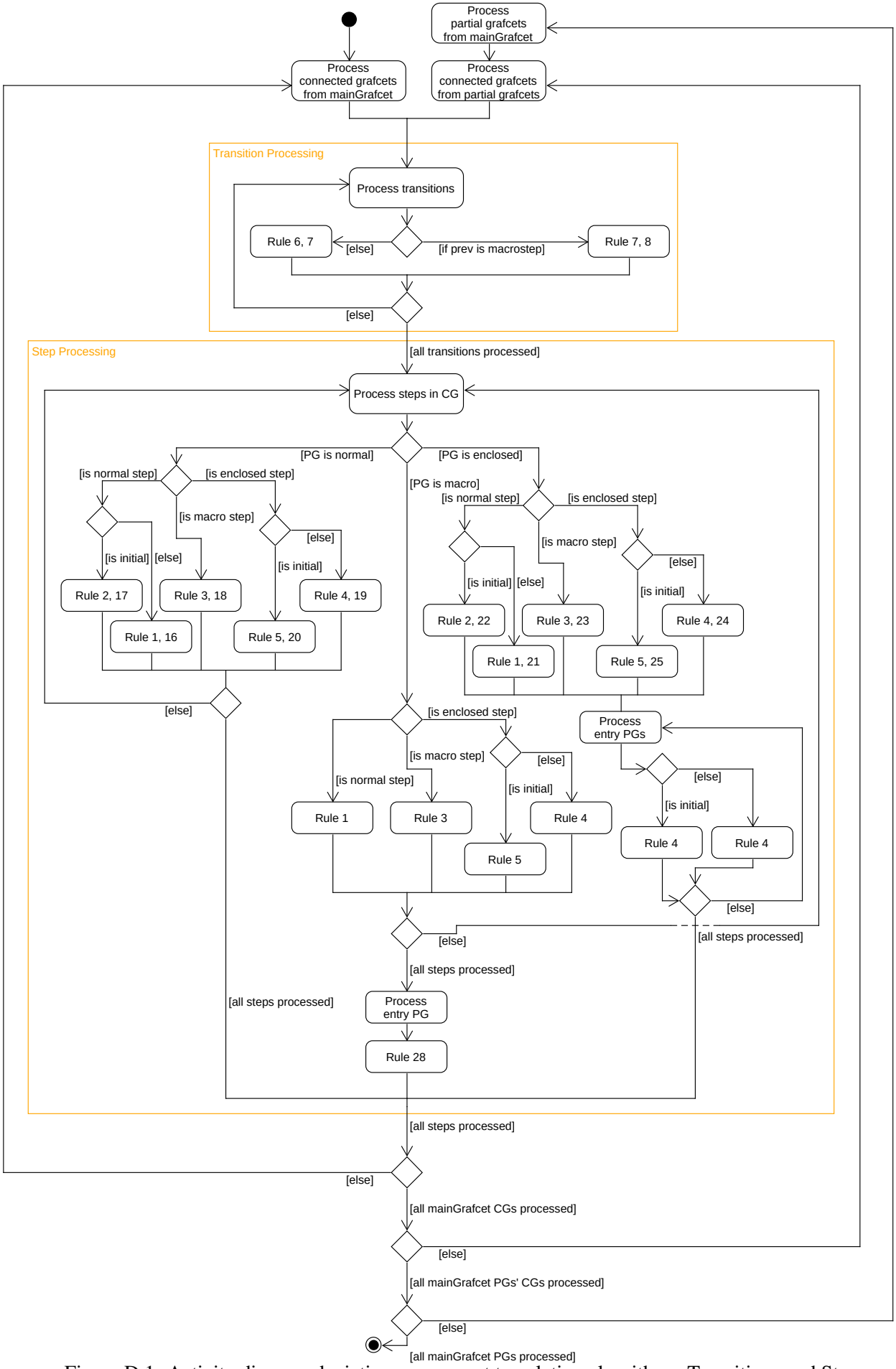


Figure D.1: Activity diagram depicting component translation algorithm - Transitions and Steps.

After having processed all the transitions and steps with the adequate translation rules and JSON model integration, the next step in the translation algorithm requires the actions to go through processing.

Instead of having the algorithm build the template for the transitions, steps and actions per connected grafcet of the main grafcet and only then address the partial grafcet connected grafcets, it first goes over all transitions and steps, as described previously, for the connected grafcets of the main grafcet and partial grafcet connected grafcets and then goes over the actions for the same grafcets. This is done in order to ensure that the actions are properly created and stored in the JSON model, as they may depend on certain variables belonging to another POU that does not include said action. These variables, and the POU itself, must exist so that the process goes through correctly, otherwise it will result in an invalid access and subsequent crash of the program.

As previously referenced, the second part of the translation algorithm, depicted in the activity diagram of figure D.2, describes the conditions in which the rules for the translation of actions are applied.

For this end, the function **build_actions_template()** is used. This function, apart from initializing some action related POU methods, begins by iterating over all steps present in the connected grafcet being analyzed and subsequently their associated action components. These actions are sorted according to their type, where the appropriate rules outlined in the aforementioned thesis are being applied.

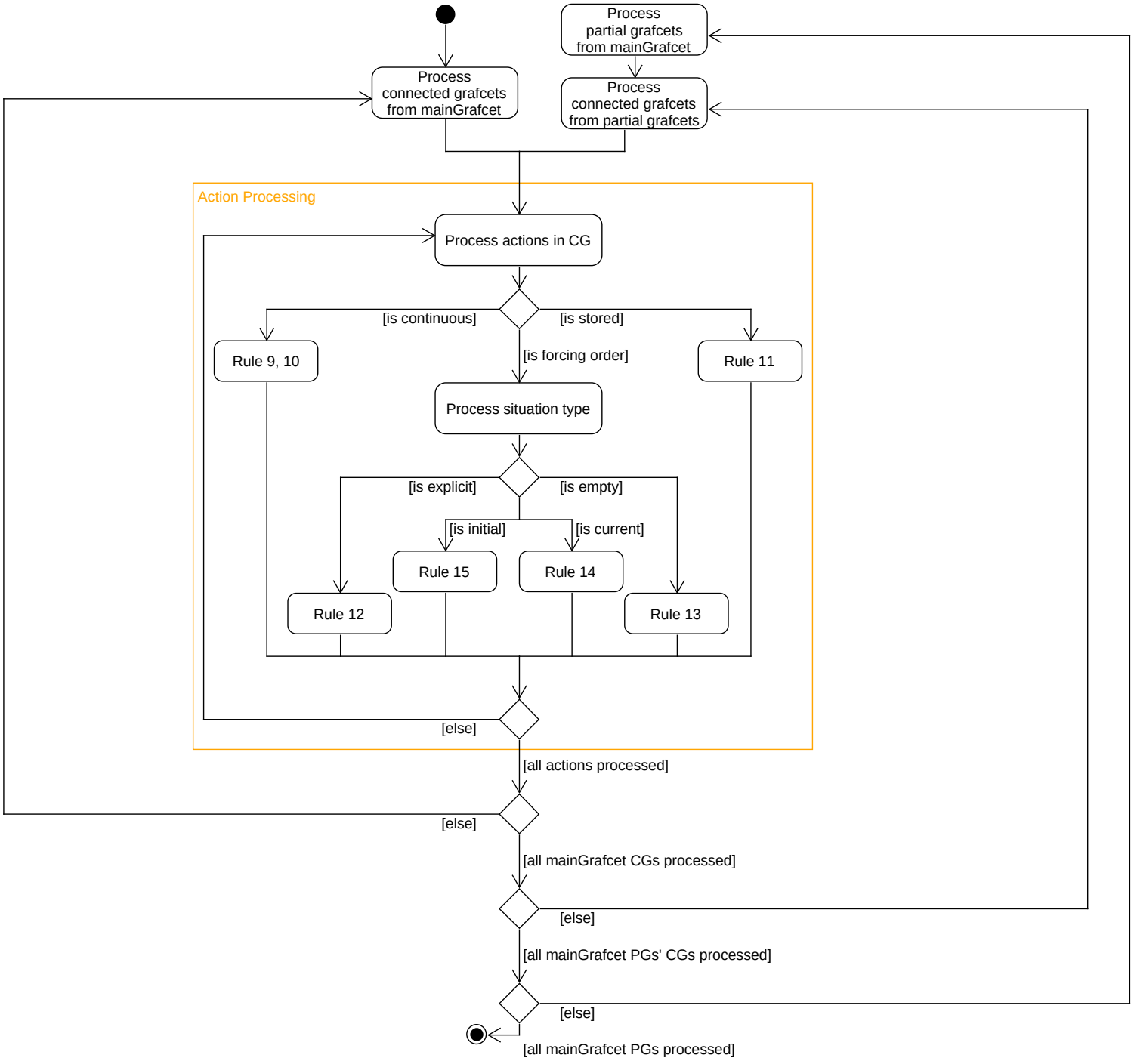


Figure D.2: Activity diagram depicting component translation algorithm - Actions.

As evident by figures D.1 and D.2, the Structured Text Translator tool begins by addressing the connected grafcets of the main grafcet, only then processing the connected grafcets belonging to the partial grafcets; it does this for both of these translation code blocks.

Note that [33] does not provide a complete rule set for processing stored and event driven actions. With there only being rule 11 present, which is related to on activation stored actions, it

has been decided to expand upon this rule. In figure D.2 the reference to rule 11 describes three separate rules, that each tackle different types of stored action. These rules are described in figure D.3, where rule 11.a addresses on activation stored actions, rule 11.b addresses on deactivation stored actions and, finally, rule 11.c addresses event triggered actions. Additionally, it provides a comment that shares insight into how trigger related variables, also used for the purposes of these actions, were handled.

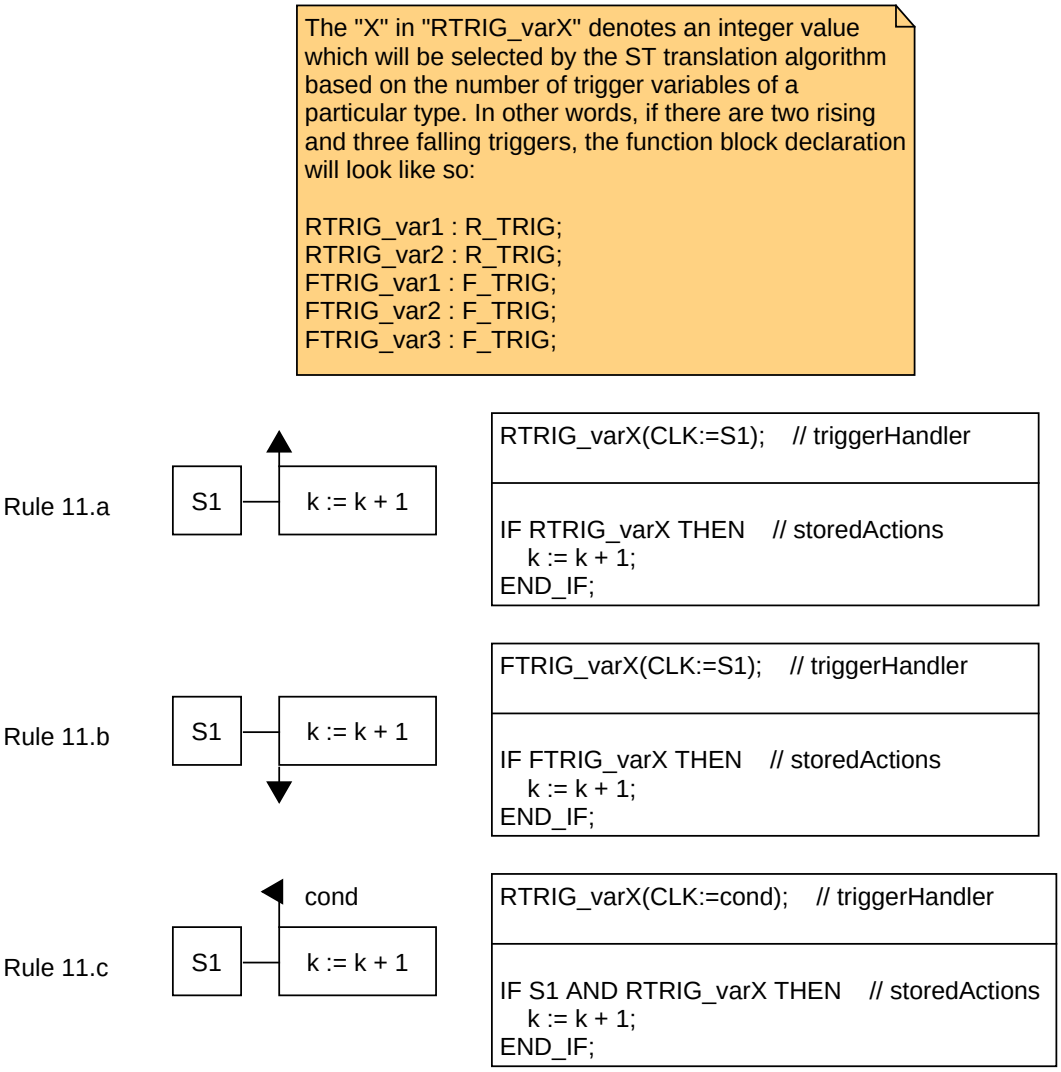


Figure D.3: Rule 11 rule set expansion.

With this the action rule processing comes to an end.