Master's Thesis

Max Schürenberg

Transformation of GRAFCET-Based Control Specifications Into an IEC 61131-3 Implementation

July 30, 2015

supervised by:
Prof. Dr. Sybille Schupp
Prof. Dr. Alexander Fay
Dr. Frank Schumacher

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Statutory Declaration

I, Max Schürenberg, declare that I have authored this thesis independently, that I have not made use of any aid other than those sources / resources acknowledged in this thesis. Neither this thesis, nor any other similar work, has been previously submitted to any examination board.

Hamburg, 29.07.2015

______________________________
Max Schürenberg
# Contents

1. **Introduction**  
   1.1. Related Work ................................................. 2  
   1.2. Outline .................................................. 3  

2. **Background**  
   2.1. Grafcet .................................................. 5  
      2.1.1. General Aspects of Grafcet .......................... 5  
      2.1.2. Hierarchical Concepts in Grafcet ................. 8  
      2.1.3. Formal Definitions of Grafcet ..................... 12  
   2.2. IEC 61131-3 ............................................. 14  
      2.2.1. General PLC Functionality and Program Structuring 15  
      2.2.2. IEC 61131-3 Languages ............................... 16  
      2.2.3. Object-Oriented Extensions ....................... 19  

3. **Concept Development for the Transformation**  
   3.1. General Program Structure ................................. 21  
   3.2. Choice of IEC 61131-3 Programming Language ............. 22  
      3.2.1. Analysis of Sequential Function Charts as a Target Language 23  
      3.2.2. Analysis of Structured Text as Target Language ....... 30  

4. **Transformation Concept**  
   4.1. Grafcet Interpretation Algorithm ......................... 33  
   4.2. Resulting Transformation Concept .......................... 38  
      4.2.1. Transformation Procedure .............................. 43  

5. **Prototypical Implementation of Transformation Concept**  
   5.1. Tool Support ............................................... 45  
      5.1.1. IEC 61131-3 Software Development Environment ........ 45  
      5.1.2. PLCopenXML ............................................ 46  
      5.1.3. Microsoft Visio Grafcet Editor ..................... 49  
      5.1.4. Read-Module of Transformation Program .............. 49  
   5.2. Implementation of Converter Class ......................... 50  
   5.3. Implementation of PLCopenXMLWriter Class ................. 53  
   5.4. Test and Outlook ........................................... 54  
      5.4.1. Events and Timed Grafcets ............................ 55  

6. **Conclusion**  
   6.1. Future Work .............................................. 58  
      6.1.1. Possibility of Bidirectional Transformation .......... 59  

A. **Appendix** ............................................... 63
List of Figures

1.1. Waterfall Model According to [40] ................................................. 1

2.1. Structure of a Basic Grafcet ..................................................... 6
2.2. Example of Transient State Evolution .......................................... 8
2.3. Grafcet with Macrostep ............................................................ 9
2.4. Grafcet with Enclosing Step / Enclosed Grafcets ............................. 10
2.5. Grafcet with Forcing Order / Forced Grafcet ................................ 11
2.6. Cyclic and Periodic Program Execution in a PLC [12] ....................... 16
2.7. Ladder Diagram Mini-Example .................................................. 17
2.8. Function Block Diagram Mini-Example ........................................ 18
2.9. Sequential Function Charts Mini-Example ..................................... 18

3.1. Simple Example for the Transformation of Forcing .......................... 25
3.2. Extended Example for the Transformation of Forcing - Grafcet ............. 26
3.3. Extended Example for the Transformation of Forcing - SFC .................. 27
3.4. Example of the Implementation of the Grafcet in Figure 2.3 in SFC .......... 28
3.5. Simple Example of the Transformation of Enclosure - SFC ................. 29

4.1. Examples for the Modeling of Concurrent Transitions ...................... 37
4.2. Example for Concurrent Actions ............................................... 38
4.3. Process Steps of Transformation Concept .................................... 43

5.1. Tools and File Formats used During Transformation ....................... 45
5.2. PLCopenXML Representation of an IEC 61131-3 POU ....................... 47
5.3. Comparison of Function-Block Implementation in Codesys and According to IEC 61131-3 ...................................................... 48
5.4. Simplified UML-Diagram of Internal Object Model .......................... 50
5.5. Simplified UML-Diagram of Extensions to the Internal Object Model .... 51
5.6. Activity Diagram of Transformation Program ................................ 52
5.7. Test Case Concept for the Combination of Hierarchical Elements .......... 54
5.8. Example of the Usage of Edge Detection Function Blocks .................. 56

A.1. Activity Diagram of Transformation Program - Subchart Process Steps .... 71
A.2. Activity Diagram of Transformation Program - Subchart Process Actions .... 72
A.3. Extended Example of a Grafcet ............................................... 73
A.4. Transformed IEC 61131-3 Control Program of the Grafcet in Figure A.3 after Codesys Import ...................................................... 75
1. Introduction

The worksteps involved in software development and systems engineering typically include the evaluation of the problem/environment, the definition of requirements, a technical analysis of the requirements, the design and development of the system followed by deployment and maintenance. These worksteps are referred to as *systems development lifecycle (SDLC)* [40]. There are numerous models how to arrange these stages, mainly characterized by a linear process (e.g., waterfall model) or iterative process (e.g., spiral model). Figure 1.1 shows an illustration of the SDLC according to the waterfall model. Today’s development of industrial automation systems is typically based on a set of requirements defined in natural language. These requirements are directly implemented in a programming language for the target architecture. Referring to the waterfall model, that means that requirement specification and analysis are typically merged into one step and undertaken by an automation engineer. The design and development are merged into yet another single step done by a technician or field engineer (control practitioners). This procedure might save a lot of documentation and coordination effort, but it is subject to human interpretation and thus very error prone. The terms used in natural language are undefined, and the expression of relationships defined by the requirements are difficult to quantify because the author often infers relationships that are implicit with the language [23]. In desktop application development, the paradigm *model-driven development (MDD)* is a significant improvement to overcome these issues [45]. The set of requirements is expressed using unambiguous models before the actual implementation of the algorithms (analysis stage). The *Unified Modeling Language (UML)* [31] is an example for such a model. Furthermore, if the model is subject to a computer-interpretable formal specification, it can be used to automate other tasks in software development, e.g., program code or test case generation.

As automation systems become increasingly complex, maturing from stand-alone
single-purpose functions to integrated parts of a fully digitized supply-chain \cite{9}, the use of model-based design is a logical and necessary step. Although widely accepted among researchers \cite{48} \cite{15} \cite{45}, most approaches towards MDD still lack industry acceptance. One main reason, among others, is that the models used in software engineering (e.g., UML) are usually not familiar to control practitioners. In the field of industrial control systems, Programmable Logic Controllers (PLC) are subject to the international industry standard IEC 61131-3 and are de-facto standard as a control hardware. The IEC 61131-3 offers five programming languages that include graphical languages to structure the control flow. However, their degree of abstraction is relatively low and not suitable to be used as a model language themselves.

Recent work on the introduction of MDD in automation solution development suggests to use Grafcet as a basic principle \cite{43}. Grafcet is a standardized graphical specification language for the design of a controller’s dynamic behavior. Since it is part of the curriculum of German control practitioners, \cite{30}, its usage is of particular interest and would overcome the obstacle mentioned above. The Grafcet standard itself states:

This specification language should also serve as a communication means between designers and users of automated systems. \cite{18, p. 7}

With the possibility of hierarchical structuring, Grafcet offers a high level of abstraction while still being close to the implementation. As such, it can serve as a convenient way of documenting the control part of an automation system. Still, control logic specified in Grafcet requires the manual implementation in one of the IEC 61131-3 languages. At the author’s institute the research efforts aim at completely removing all manual impact from this process by developing an automated transformation of Grafcet specifications into an IEC 61131-3 implementation. Referring to the waterfall model, the interface between the design and development stage shall ideally be supported by a ‘one-click-solution’. At the current stage, a transformation algorithm has been developed that transfers a Grafcet into an equivalent control interpreted Petri net \cite{11} and transforms it into the IEC 61131-3 programming language Sequential Function Charts. The disadvantage of that transformation algorithm is the necessity to normalize a Grafcet, i.e., replace all hierarchical elements by equivalent basic constructs \cite{42}. The possibility of hierarchical structuring is one of the main powers of Grafcet, so a different transformation approach is desirable. The aim of this thesis is to find a transformation algorithm of a Grafcet specification into an IEC 61131-3 implementation that preserves the initial hierarchical structure.

1.1. Related Work

The context of this thesis can be described as a further development of the work on \cite{42} by Schumacher. The author aims at an automated approach to generate an IEC 61131-3 implementation from a Grafcet specification. Control interpreted Petri nets (CIPN)\cite{11} serve as a formal basis for the transformation. CIPN do not include the definition of hierarchical elements. Using an approach called normalization, the hierarchical elements in
Grafcet are transformed into corresponding constructs consisting only of basic elements. The result is one single Grafcet chart that can be transformed to an equivalent IEC 61131-3 control program in the Sequential Function Chart language (SFC) using transformation rules. One main power of Grafcet is the possibility to hierarchically structure a control program. This facilitates the modeling of concurrent behavior and increases readability of the specification. The drawback of the transformation approach is that the hierarchical structure is removed, resulting in relatively large and hard-to-read SFC control programs. The latter is the main motivation for further considerations as to if it is possible to find a transformation approach that preserves the hierarchical structure. This, in turn, resulted in the project definition for this thesis. Our work also contains the development of a prototype that shall serve as a proof-of-concept for our transformation approach. This prototype is based on the tool chain used and developed in [42]. Our tool chain is partly the result of a number of other theses, i.e., the development of a Grafcet editor [41] and the transformation of the data from that editor to a Grafcet meta-model [34].

An important aspect of the transformation is to understand the semantics of both Grafcet and the IEC 61131-3 language of choice. Bauer dedicates her dissertation [5] to the development of a unifying semantics of Sequential Function Charts, which also play a key role in this thesis.

Numerous approaches exist to translate Grafcet into a formal model and/or a control program. In [39], Provost et al. translate a basic Grafcet specification into Mealy machines, [8] is an earlier work that describes the Grafcet semantics with transition systems. The authors of [7] aim at using Grafcet for the specification of C-programmed microcontrollers with an automated approach of translating a Grafcet into equivalent C-code. Apart from the representation of Grafcet elements in corresponding C-code fragments, a key aspect of [7] is to find a way of implementing a Grafcet specification on a sequential single CPU system. This results in the definition of a Grafcet interpretation algorithm. The resulting algorithm does not include hierarchical elements and may also lead to false execution of initial actions. However, the overall structure of their considerations is in close relation to this thesis.

In [11], David and Alla deal with control interpreted Petri nets and include an interpretation algorithm. In the appendix the authors state that CIPN can be interpreted as a Grafcet and vice versa. For future work they suggest to use their CIPN interpretation algorithm, extend it by the notion of hierarchical elements and use it for the transformation of Grafcet into an IEC 61131-3 implementation. This is exactly what we have done in this thesis.

1.2. Outline

This work is organized as follows: In the chapter Background, we give an overview of the characteristics of Grafcet and the content of the IEC 61131-3, as this thesis builds on these two bodies of knowledge. In Concept Development for the Transformation, we analyze the available IEC 61131-3 programming languages for their suitability to be a
target language of the transformation. We also include a discussion of possible program structures of the transformed IEC 61131-3 implementation. This chapter forms a basis for the Transformation Concept. We develop an interpretation algorithm for Grafcet and define transformation rules that map a specific set of IEC 61131-3 code fragments to each element of Grafcet. Our work also includes the implementation of the transformation rules as a proof of concept which we explain in the chapter Prototypical Implementation of Transformation Concept. Ending with the Conclusion, the thesis also provides a summary as well as hints for future work.
2. Background

To fully understand the challenges and limiting factors of the transformation of Grafcet into an IEC 61131-3 implementation, it is important to have a solid understanding of both Grafcet and the content of the IEC 61131-3. Section 2.1 explains the structure and dynamic behavior of Grafcet and defines a formal definition that is the basis of the transformation program. Section 2.2 contains the most important aspects of the IEC 61131-3 industrial standard, including a brief overview of its programming languages.

2.1. Grafcet

Grafcet is a standardized graphical specification language used to describe the behavior of logical control systems. Its origins trace back to the 1970s, when the advent of Programmable Logic Controllers (PLCs) introduced software-based control logic that replaced the physical wiring of relay systems [22]. The first programming languages included assembler-like languages as well as graphical means of virtually designing the hardware wiring of the system. Instruction List and Ladder Diagram are remnants of such languages in today’s PLC programming standard IEC 61131-3 (see section 2.2). In 1977, the French Association Francaise pour la Cybernétique Economique et Technique (AFCET) aimed at reducing the learning curve for PLC users and finding a specification language that reflects the control flow rather than a particular implementation technology. The resulting specification language was inspired by Petri nets [11] and called Graphe Fonctionnel de Commande Etape Transition (GRAFCET) [10]. It was first recognized as the international norm IEC 848 in 1988 [19]. Its current version is edition 3 from 2013, now under the number IEC 60848 [18]. The author of [42] gives an overview of the historical development and the contributions to the norm.

2.1.1. General Aspects of Grafcet

A Grafcet describes the control part of a sequential and concurrent automation system. In essence, this control part indicates how the output variables of a system depend on its input variables and its internal state. Grafcet is a step-transition system much like Petri nets. The representation of a control part in Grafcet distinguishes between the structure, which characterizes possible states of the automation system, and the interpretation, which defines the actual input / output relations and possible system state evolutions.

Structure

The structure of a Grafcet is characterized by two disjunct sets of nodes: *steps*, graphically represented by squares, and *transitions*, represented by horizontal bold lines. These nodes are alternately connected by directed links, i.e., two steps or two transitions can not be directly connected with each other. Steps and transitions each have unique names.
2. Background

to enable their unambiguous identification. In the example in figure 2.1 the numbers in the step symbols are the step names, while the values inside brackets next to the transitions are the transition names. If more than one step precedes or succeeds a transition, a horizontal double line called *synchronization* is used. A step can be either active or inactive, which is indicated by the boolean step activity variable that returns *true* if the step is active and *false* otherwise. An active step is marked with a token in the Grafcet chart. The set of all active steps at any given time instant is called the *situation* of the Grafcet. Referring to the example again, the situation at the current time \( m \) is \( \text{sit}(m) = [21, 24] \). The situation on initialization of the Grafcet is given by the set of initial steps, denoted by steps with a double frame. In the example, this means \( \text{sit}(\text{init}) = [21, 22] \). Further Grafcet syntax elements are explained in the following sections.

![Figure 2.1.: Structure of a Basic Grafcet](image)

**Interpretation**

The interpretation of a Grafcet requires elements that relate the system’s inputs and outputs to the internal state evolution. These elements are *transition conditions* and *actions*. Each transition has an associated transition condition, written next to the transition symbol, which is a logical expression consisting of input and internal variables. A transition condition evaluates to *true* or *false*, depending on the logical combination of its variables. A transition condition can also contain events, denoted by an up-arrow for the rising and a down-arrow for the falling edge of a signal. For example, the transition condition \( t_1 = \uparrow a \) evaluates to *true* if the signal \( a \) changes its value from *false* to *true*. \( t_1 \) is set back to *false* immediately after occurrence of the event. The transition conditions determine the evolution of the system as described in the next section.

Actions define the manipulation of output and internal variables by the Grafcet depending on the current situation. An action is represented by a rectangle that is connected to a step by a horizontal line. This indicates the logical connection between the
execution of the action and the activity of the corresponding step, e.g., execution only while the step is active. There are two kinds of actions: A continuous action indicates that an output signal has a true value when the corresponding step is active. Globally, the value of an output signal is determined by an OR operation on all corresponding continuous actions. This means that these signals are false if no assigned action is active. Stored actions set an internal or output variable to a dedicated value on activation / deactivation of the corresponding step. The variable maintains this value until another stored action resets it. A stored action is marked by an arrow at the left edge of the action symbol. The Grafcet in figure 2.2 contains both continuous and stored actions. On activation of step 4, the value of k is set to 5. It will keep this value until the next activation of step 5, when k is set back to 1. The variable signal1 is only true while step 5 is active and false otherwise. The same applies to signal2 and step 6.

Dynamic Behavior

The dynamic behavior of a Grafcet, i.e., the evolution from one active situation to the next, is given by five evolution rules [18, p. 14].

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 All 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.

In this work, clearing and firing of a transition are synonymous.

Particular attention has to be put on the combination of rule 3 and 4. If several subsequent transitions are all enabled on occurrence of an event, they all clear simultaneously. The next situation is called stable situation and is given by the set of steps that has been activated by the last transition. All intermediate steps are part of an unstable situation, which has only virtually been activated and deactivated. This behavior is called transient evolution. During transient evolution, continuous actions that belong to steps of an unstable situation are not executed. In contrast, all stored actions are executed, no matter if their steps belong to a stable or unstable situation.

Example 2.1.1. Consider figure 2.2. Assume that in the current situation sit(m) = [3], the boolean variables e and f are true, while the boolean variables d and g are false. This means that only transitions t12 and t13 are enabled. The stored action associated to step 3 assigns the value 1 to k. If the variable d changes its value from
2. Background

false to true, the next situation at timestep $m+1$ is $\text{sit}(m+1) = [6]$, and the steps 4 and 5 were ‘skipped’. Nevertheless, $k$ has changed its value from 1 to 5 because of the stored action associated to step 4. At the stable situation $\text{sit}(m+1)$, $\text{signal2}$ is true. The variable $\text{signal1}$, however, has never been true because the continuous action was part of the unstable situation.

The evolution rules guarantee deterministic behavior\[10\] and allow for a simulation of the control program as a token play. Yet, if the Grafcet shall be interpreted on a sequential process such as a CPU in a PLC, additional information on the orders of rule 2, 3, and 4 is needed. A possible implementation of such an order is the basis of section 4.1.

2.1.2. Hierarchical Concepts in Grafcet

To improve readability of the dynamic behavior of complex automated systems and allow the modeling of concurrent entities, the IEC 60848 offers three different ways of structuring a Grafcet and introducing hierarchy levels to the specification. Grafcet allows the composition of several Grafcets, so-called partial-Grafcets into one global Grafcet. In the application domain, these partial-Grafcets could represent the different operating modes of the automated system, while the global Grafcet represents the high-level control-flow. The control-flow between the several Grafcets is realized by the different means of hierarchical structuring. The IEC 60848 has evolved over time and did not contain the possibility for hierarchical structuring in its early versions. The following descriptions of Grafcet-elements follow the latest version of the IEC 60848 \[18\].
A macrostep serves as a placeholder of a part of the specification and allows the composition and decomposition of a Grafcet. It is represented graphically by the step symbol with two additional horizontal lines at the top and bottom and the prefix M in its name. Because of its placeholder function, no action should be associated to a macrostep. The detailed part of the specification that is replaced by the macrostep is called expansion of the macrostep and is specified in a separate partial-Grafcet. An expansion of a macrostep has to consist of one entry step with prefix E and one exit step with prefix S (abbreviation for French: Entrée/Sortie). A macrostep expansion can only belong to one macrostep, though this macrostep can be used several times in the Grafcet. If more than one instance of a macrostep can be simultaneously active, the implementation requires several copies of the expansion respectively. The entry step becomes active once the macrostep becomes active, too. The expansion is then executed and the macrostep remains active until the exit step of the expansion is reached. The succeeding transition of the macrostep can not be enabled until the exit step of the expansion becomes active. It is possible to use a macrostep without a corresponding expansion. In that case, the macrostep behaves like a normal step.

Example 2.1.2. Figure 2.3 shows an example of the usage of macrosteps in a Grafcet. The Grafcet on the left represents the high level of an automated process that makes use of the macrostep $M_2$. The right partial-Grafcet is the expansion of the macrostep $M_2$. The suffix of the entry step $E$ and the exit step $S$ and the label on the frame of the partial-Grafcet are distinct references to the macrostep $M_2$. In the current situation of the example, a value change of the variable $a$ from false to true leads to the firing of transition $t_1$, which deactivates step 1 and activates $M_2$ and thus $E_2$. The macrostep expansion is now active. The transition $t_2$ cannot be enabled until step $S_2$ becomes active, e.g., by successive clearing of transition $t_{20}$ and $t_{22}$. When $S_2$ is active and $b$ is true, transition $t_2$ can clear and deactivate $S_2$ and thus $M_2$.

Macrosteps offer a way to ease modeling and improve the understanding of a Grafcet.
High-level systems can be specified by using empty macrosteps in the beginning. In later stages of the engineering process, functionality can be added by specifying the corresponding macrostep expansions.

**Enclosing Steps**

Enclosing steps offer a way of implementing a hierarchical structure between autonomous partial-Grafcets. The activation of an enclosing step of a higher-level partial-Grafcet simultaneously activates specific steps of a lower-level partial-Grafcet, referred to as partial-Grafcet of enclosing steps or enclosed partial-Grafcet. The partial-Grafcet of enclosing steps follows the five evolution rules until the enclosing step gets deactivated again. The enclosing step is represented by a step-symbol with a diagonal line in each of its corners. The steps of the enclosed partial-Grafcet that are activated with the enclosing step are marked with an asterisk next to the step-symbol. A partial-Grafcet of enclosing steps contains a distinct reference to its enclosing step. The deactivation of an enclosing step leads to the deactivation of all steps of the partial-Grafcet of enclosing steps. The relation of an enclosing step to partial-Grafcets of enclosing steps is 1:n.

![Figure 2.4: Grafcet with Enclosing Step / Enclosed Grafcets](image)

**Example 2.1.3.** In figure 2.4 the left partial-Grafcet \textit{GMain} represents the highest hierarchy level. The activation of step 2 leads to the activation of the steps 22 and 32 of the two enclosed Grafcets \textit{GWeighing} and \textit{GMixing}. The labels on their frames show the names and the reference to the enclosing step 2. All three partial-Grafcets operate under the evolution rules until the clearing of transition \textit{t2}. This deactivates step 2 and thus all steps of \textit{GWeighing} and \textit{GMixing}.

**Forcing Orders**

Forcing orders are another way of controlling lower-level partial-Grafcets from high-level charts. They offer the possibility of forcing a partial-Grafcet to a specific situation. Forc-
2.1. Grafcet

Forcing orders are embedded in a double rectangle and associated to steps with a horizontal line. Table 2.1 shows the syntax of forcing orders. The forcing order is termed *internal order* and has priority over the evolution rules. That means that a forced Grafcet cannot evolve until the forcing order has finished, i.e., the associated steps get deactivated. The forced Grafcet is said to be frozen. If the forcing order is not active, the Grafcet continues to operate under the evolution rules. The relation of forcing orders to forced Grafcets is $m:1$, which means that a forced Grafcet can be forced from several different forcing orders, as long as these orders are not active at the same time. A step can have more than one forcing order. A forcing order has the same dynamic properties as a continuous action, thus it is not executed when its step is part of an unstable situation.

<table>
<thead>
<tr>
<th>Forcing Order</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G3{8,9,11}</td>
<td>Forcing the partial-Grafcet G3 into the situation where only the steps 8, 9, 11 are active</td>
</tr>
<tr>
<td>G4{INIT}</td>
<td>Forcing of G4 into the initial situation</td>
</tr>
<tr>
<td>G5{*}</td>
<td>Forcing the current situation of G5</td>
</tr>
<tr>
<td>G6{}</td>
<td>Forcing G6 to the empty situation (no step is active)</td>
</tr>
</tbody>
</table>

Table 2.1.: Syntax for Forcing Orders in Grafcet

Example 2.1.4. Figure 2.5 illustrates the case where a partial-Grafcet is forced to a specific situation. Both Grafcets are in their initial situation and apply the evolution rules in parallel, e.g., the clearing of transition $t_{20}$ deactivates step 20 and activates the steps 21 and 22 of G1. Clearing transition $t_1$ activates step 2, which enables the forcing...
Table 2.2.: Dynamic Properties of Hierarchical Elements in Grafcet

<table>
<thead>
<tr>
<th>Property</th>
<th>Macrostep</th>
<th>Enclosure</th>
<th>Forcing Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior of subchart on activation of hierarchical element</td>
<td>Activates entry step E*</td>
<td>Activates enclosed steps</td>
<td>Activates forced situation</td>
</tr>
<tr>
<td>Behavior of subchart on deactivation of hierarchical element</td>
<td>Deactivates all steps</td>
<td>Deactivates all steps</td>
<td>Continues evolution under evolution rules</td>
</tr>
<tr>
<td>Behavior of main chart while hierarchical element is active</td>
<td>Macrostep stays active until exit step of expansion is reached</td>
<td>Continues evolution under evolution rules</td>
<td>Continues evolution under evolution rules</td>
</tr>
</tbody>
</table>

Summary of Dynamic Behavior for Hierarchical Elements

The basis for the transformation rules for translating Grafcet elements into IEC-61131-3-structures is the dynamic behavior. Ensuring a correct transformation means ensuring equivalent dynamic behavior. Each of the structure presented in this chapter has a different impact on the evolution of the high- and low-level partial-Grafcets. The comparison of these properties in table 2.2 allows distinguishing the hierarchical structures of Grafcet.

2.1.3. Formal Definitions of Grafcet

To be able to transform Grafcet into an IEC 61131-3 implementation, or in fact into any other representation, it is necessary to have a formal background of the Grafcet syntax. On that basis, a transformation algorithm is able to identify the Grafcet elements and transform them. We developed the following definitions, based on [38] and extended by the notions of enclosures and forcing orders. We shall use these definitions for the development of a Grafcet interpretation algorithm for sequential systems in section 4.1 as well as for the construction of a prototypical transformation program in section 5.2.

Definition 2.1.1. A Grafcet G is a 4-tuple \((I_G, O_G, C_G, S_{initG})\) where
- \(I_G\) is the non-empty set of logic inputs
- \(O_G\) is the non-empty set of logic outputs
- \(C_G\) is the set of Grafcet charts
2.1. Grafcet

- $S_{\text{InitG}}$ is the set of initial steps

As table 2.2 illustrates, macrostep expansions and enclosed partial-Grafcets are different from other charts in the way that they contain a set of ‘special steps’. Thus, the charts set is divided into the set $C_C$ of classical charts, the set $C_M$ of macrostep expansion charts, and the set $C_E$ of enclosed partial-Grafcets.

**Definition 2.1.2.** A classical chart $c \in C_C$ is defined by a 4-tuple $(m, S, T, A)$ where
- $m$ is the name of the chart
- $S$ is the non-empty set of steps of $c$
- $T$ is the set of transitions of $c$
- $A$ is the set of actions of $c$

**Definition 2.1.3.** A macrostep expansion chart $cm \in C_M$ is defined by a 6-tuple $(m, s_I, s_O, S_{\text{oth}}, T, A)$ where
- $m$ is the macro-step name
- $s_I$ is the input step of the expansion
- $s_O$ is the output step of the expansion
- $S_{\text{oth}}$ is the set of other steps of $cm$
- $T$ is the set of transitions of $cm$
- $A$ is the set of actions of $cm$

**Definition 2.1.4.** An enclosed partial-Grafcet (or partial-Grafcet of enclosed steps) $enc \in C_E$ is defined by a 5-tuple $(xm, S_{\text{enc}}, S_{\text{oth}}, T, A)$ where
- $xm$ is the name of the enclosure
- $S_{\text{enc}}$ is the non-empty set of enclosed steps $sx$ in $enc$
- $S_{\text{oth}}$ is the set of other steps of $enc$
- $T$ is the set of transitions of $enc$
- $A$ is the set of actions of $enc$

**Definition 2.1.5.** The step sets are divided into the sets of normal steps $S_s$, the sets of macrosteps $S_m$ and the sets of enclosures $S_e$. All steps $s \in (S_s \cup S_m \cup S_e)$ are 3-tuples $(m, x, t)$ where
- $m$ is the name of $s$
- $x$ is a boolean variable indicating if $s$ is active or not
- $t$ is a timer variable indicating the active duration of $s$

Let $S(cm)$ be the set of all steps of the macrostep expansion chart $cm$ ($S(cm) = \{s_I, s_O\} \cup S_{\text{oth}}$), $S(enc)$ be the set of all steps of the enclosed partial-Grafcet $enc$ ($S(enc) = S_{\text{enc}} \cup S_{\text{oth}}$) and $S_G$ be the set of all steps $s$ of the Grafcet.

**Definition 2.1.6.** A transition $t \in T$ of a chart $c$ is defined by a 3-tuple $(S_B, S_A, E_{\text{Cond}(I_G, S_G)})$ where
- $S_B$ is the set of immediate predecessor steps of $t$ (before)
- $S_A$ is the set of immediate successor steps of $t$ (after)
- $E_{\text{Cond}(I_G, S_G)}$ is a transition condition. It is a boolean expression of inputs and step activity variables
Let \( T_G \) be the set of all transitions \( t \) of a Grafcet.

The set of actions \( A \) is divided into the set \( A_S \) of stored actions with a set of associated outputs \( O_S \), the set \( A_C \) of continuous actions with outputs \( O_C \) and the set \( A_F \) of forcing orders. The union of \( O_S \) and \( O_G \) forms the output set \( O_G = O_S \cup O_C \).

Let \( A_G \) be the set of all actions \( a \) of a Grafcet. Let \( A_{GS}, A_{GC} \) and \( A_{GF} \) be the sets of all stored actions \( a_S \), all continuous actions \( a_C \) and all forcing orders \( a_F \) of a Grafcet accordingly.

**Definition 2.1.7.** A continuous action \( a_C \in A_C \) of a chart \( c \) is defined by a 3-tuple \((s, o, E_{\text{Cond}(I_G, S_G)})\) where
- \( s \) is the step to which the action is connected
- \( o \) is the output which is manipulated by the action
- \( E_{\text{Cond}(I_G, S_G)} \) is a continuous action condition, consisting of a boolean expression on inputs and step activity variables

**Definition 2.1.8.** A stored action \( a_S \in A_S \) of a chart \( c \) is defined by a 4-tuple \((s, o, \text{val}, \text{inst})\) where
- \( s \) is the step to which the action is connected
- \( o \) is the output which is allocated by the action
- \( \text{val} \) is the value that is assigned to the output
- \( \text{inst} \) is the instant when the allocation is done:
  \[ \text{inst} \in \{\text{StepActivation, StepDeactivation}\} \]

**Definition 2.1.9.** A forcing order \( a_F \in A_F \) of a chart \( c \) is a 4-tuple \((s, c, op, sit)\) where
- \( s \) is the step to which the action is connected
- \( c \) is the forced partial-Grafcet from the set of classical charts, \( c \in C_C \)
- \( op \) is the kind of forcing operation: \( op \in \{\text{Spec, Init, Current, Empty}\} \), where
  - \( \text{Spec} \) is a specific situation of the forced partial-Grafcet,
  - \( \text{Init} \) the initial situation,
  - \( \text{Current} \) the current situation and \( \text{Empty} \) the empty situation where no step is active
- \( sit \) is the set of steps \( s \in S \) of the forced partial-Grafcet that marks the forced situation

### 2.2. IEC 61131-3

The IEC 61131-3 is a part of the IEC 61131 international standard. The IEC 61131 consists of 8 different parts that deal with different aspects and requirements of PLC systems. The third part, i.e., the IEC 61131-3, serves as a standard for PLC programming. The most current edition is the third revision of the norm and has been released in 2013 [20]. Compared to its predecessor, it includes a major new aspect regarding the object-oriented programming of PLC systems. Since the norm has been developed by the International Electrotechnical Commission with the contribution of major PLC vendors and automation research institutions, it is widely accepted and de-facto standard
within the automation industry\textsuperscript{[49]}. Its compliance is controlled by a consortium with strong industry background, the PLCopen committee (see section 5.1.2 on page 46).

The IEC 61131-3 describes five different languages that can be used to program a PLC. These include two textual languages (Instruction List (IL) and Structured Text (ST)), as well as three graphical languages (Function Block Diagram (FBD), Ladder Diagram (LD), and Sequential Function Charts (SFC)). The general means of structuring a PLC program as well as the language-independent elements like variables are also part of the standard.

\subsection*{2.2.1. General PLC Functionality and Program Structuring}

The core of a PLC is a signal processing unit, where the execution of a control program connects input signals and internal variables to output signals or other internal variables. The processing part of a PLC typically works in a cyclic or periodic way, as illustrated in figure 2.6. After reading the inputs, the control program is executed and the outputs are written after its termination. This principle originates from the initial hard-wiring of control logic in early automation control systems.

According to the IEC 61131-3 standard, a PLC system itself is called a \textit{configuration}. It may consist of one or more CPUs, which are referred to as \textit{resources}. A resource consists of a set of global variables and of \textit{task} declarations that assign run-time properties to the actual control program elements described below. These run-time properties include the cycle time in which the associated program elements are called and a priority flag that compares the priority of the associated element to other elements running concurrently. The function of the latter is implementation-dependent. Systems may either implement preemptive scheduling, i.e., a task with lower priority is interrupted immediately, or non-preemptive scheduling, i.e., the resource waits for termination of the task with lower priority until tasks with higher priority may start \textsuperscript{[22]}. This behavior is an important consideration in section 3.1.

A PLC control program consists of a collection of functionally independent software units, called program organization unit (POU). The norm defines three different types of POUs, which have different characteristics: \textit{functions}, \textit{function blocks}, and \textit{programs}.

A function can have an arbitrary number of input values and exactly one output value. The property of a function is that it always gives the same result for the same input values, regardless of the system’s state. As such, it does not have any internal memory, is not allowed to read any global variables and can only call other POUs of type function. An example of a function is the provision of a standard mathematical operation, e.g., root\textsuperscript{$(x, y)$}.

Programs and function blocks used to have only minor differences until the introduction of object-oriented extensions for function blocks in the latest edition of the IEC 61131-3. Both POUs can have an arbitrary number of input and output values and can save local data over several invocations, i.e., have internal memory. The main difference between programs and function blocks is that function blocks have to be instantiated to be called by other function blocks and programs. Each instance represents its own copy of the POU in the PLC memory, which means that several instances of the same
function block can be used independently. A program cannot be instantiated and can only be called by a task inside the PLC configuration.

2.2.2. IEC 61131-3 Languages

As mentioned before, the IEC 61131-3 defines five programming languages for PLC control programming. Their origins are from very different domains, e.g., hardware wiring, Petri nets or the Pascal programming language. We only present the rudimental characteristics of each language - section 3.2 on page 22 gives a more detailed view on the characteristics of the programming languages in the context of this work. Each language is illustrated with a mini-example. In the mini-examples, a boolean input variable \texttt{input1} is loaded. If it is \texttt{true}, the value of an internal counter variable \texttt{count1} is increased, otherwise it is decreased. Note that the examples have a descriptive character and might not always make sense from an implementation point of view.

\textbf{Instruction List (IL)}

Instruction List is an assembler-like programming language. IL often serves as an intermediate language in the compilation of the other four IEC 61131-3 programming languages\cite{22}. Each IL instruction consists of exactly one line. The instructions are executed one-by-one with possible jumps in-between. An instruction consists of an optional label to reach the instruction with a jump, an IL operator, and an operand, i.e., a constant or a variable that is manipulated by the operator function.

\begin{verbatim}
LOAD:  LD input1
       JMPC INCR

DECR:  LD counter1
       SUB 1
\end{verbatim}
Structured Text (ST)

Structured Text is a Pascal-like high-level programming language. As such, it includes loops, mathematical functions, iteration, and conditional execution. Its higher abstraction level compared to IL offers better readability and more intuitive programming to today’s users that are usually familiar with at least one C-like language. The downside of compiled ST code is that it is usually slower than a direct implementation in IL. However, this issue can be neglected on modern PLC systems because of fast execution times.

```plaintext
IF input1 THEN
  counter1 := counter1 + 1;
ELSE
  counter1 := counter1 - 1;
END_IF;
```

Listing 2.2: Structured Text Mini-Example

Ladder Diagram (LD)

Ladder Diagram is a graphical language based on electromechanical relay systems. Two vertical lines representing power rails are connected by rungs which embed graphical symbols that correspond to contacts, coils and function blocks. The left power rail has the logic state 1, leading to a logical flow from left to right. The resulting LD network is executed from top to bottom.

![Ladder Diagram Mini-Example](image)

Figure 2.7.: Ladder Diagram Mini-Example
Function Block Diagram (FBD)

Function Block Diagram is a graphical language that reflects the basic idea of function-oriented sequence chains. FBD is a network graphic, where rectangular boxes that represent an operation on their input variables are connected by horizontal and vertical lines that represent data flow and can pass any IEC 61131-3 variable type. A FBD diagram is usually evaluated top-to-bottom, although vendor-dependent implementations (e.g., via the assignment of sequence numbers) are possible. It is possible to directly transform a LD control program into a FBD representation because both languages share the element wiring concept.

![Function Block Diagram Mini-Example](image)

Figure 2.8.: Function Block Diagram Mini-Example

Sequential Function Charts (SFC)

Sequential Function Charts are mainly used to describe the control flow of a program. SFC is a classical Petri-net like step / transition system that is said to be based on Grafcet [22]. The entire program is broken down into smaller units and the control flow between them is controlled via conditional transition firings. The use of SFC facilitates the design of parallel processes. The small units themselves are usually programmed with the other four languages or using SFC again.

![Sequential Function Charts Mini-Example](image)

Figure 2.9.: Sequential Function Charts Mini-Example
2.2.3. Object-Oriented Extensions

A major change in the current and third revision of the IEC 61131-3 is the introduction of object-oriented programming extensions. While in desktop application development the use of object-oriented programming (OOP) with languages like C++ and Java is common, the approach is relatively new to the world of PLC programming. Nevertheless, the introduction of OOP to the IEC 61131-3 received great endorsement, offering improved reusability [49], the usage of software product lines for control programs [32], and an extensive usage of UML class diagrams [31] in IEC 61131-3 development [50]. The new language elements include:

- A fourth POU of type \textit{class}, which follows the class concept of the known OOP languages. A class includes methods and an internal and external data structure. A class can implement an interface with method prototypes. Subclasses can inherit from a parent class and extend or overwrite its methods and data structure. To use a class inside a program, it has to be instantiated (i.e., an object has to be created).

- An interface that serves as an abstraction for a class or a function block.

- Object-oriented extensions of function blocks, i.e., the inclusion of methods and the possibility to implement interfaces and inherit and extend from other function blocks.

Figure 2.10 shows how these new language elements interact. Early drafts of the new

![Figure 2.10.: Overview of Inheritance and Interface Implementation][20] p. 119]
POU type class. In our work, we will thus focus on the object-oriented extensions for function blocks, even if a class-implementation seems equivalent or even superior.

Before the introduction of these extensions, a function block was limited to only one procedural operation. If a function block had to perform different tasks, e.g., initialization and different modes of operation, it had to be called with a specific set of input values that could be used inside the function block for a case-selection. With the introduction of methods, these different tasks can be separated. Apart from the obvious advantage of better structuring, it also significantly improves readability because a task can (ideally) be clearly identified by the name of its method. This is particularly interesting for our transformation concept, which we will point out in section 3.2.2.

For a discussion of the benefits and use-cases for the other new OOP language elements of the IEC 61131-3, refer to the sources cited in this section. We will point out some potentials for our concept in the outlook section 6.1.
3. Concept Development for the Transformation

The general aim of this thesis is to transform a Grafcet specification into an IEC 61131-3 implementation. The latter term ‘IEC 61131-3 implementation’ is very vague and needs to be specified before a transformation concept can be developed. First thoughts need to consider a target architecture. The concept of configurations, tasks and POU\(\text{s}\) offers a great number of possible ways to structure a control program hierarchically. Section 3.1 discusses possible implementations. Furthermore, it is neither possible nor intended to simply translate one IEC 61131-3 language into another [35]. Thus, a transformation concept needs to focus on one specific target language. A discussion of possible target languages and a final decision is given in section 3.2.

3.1. General Program Structure

The final transformation concept is based on some general considerations regarding the structure of the control program on the PLC.

One decision that has to be made is the choice of the target configuration. For the sake of simplicity, we decided to use a simple configuration with only one resource. Such a program will also run on more advanced systems, which is not the case in the contrary way. Keeping the property of forced Grafcets that ‘run in parallel to the main Grafcet’ in mind, a first idea was to exploit the possibility of assigning program elements to different tasks that run concurrently with different priorities. The program elements in the tasks could exchange information and start / stop each other via global variables because these are declared on resource level. However, the system-dependent difference between preemptive and non-preemptive scheduling does not allow for a universal transformation approach. Such a concept would always be limited to a certain type of PLC.

The above considerations lead to a target architecture of one configuration with one resource that includes only one task that controls only one main program.

For the overall aim to keep the hierarchical structure with partial-Grafcets, we have to encapsulate the transformed IEC 61131-3 implementation of each partial-Grafcet into its own POU. The decision for a POU type can be done by exclusion:

- A POU of type \textit{function} is not applicable because it cannot save state information, which is the most essential part of Grafcet.

- A \textit{program} can only be called by its task, but not by any other POU. Because partial-Grafcets exchange information via the implicit links of hierarchical elements, the resulting POU\(\text{s}\) need to be able to call each other. A workaround with the usage of different tasks and global communication variables has been considered impractical above, so programs are not applicable either.

- The remaining POU\(\text{s}\) of type \textit{function block} and \textit{class} both suit the needs of the transformation very well. They can save state information and are able to call each
3. Concept Development for the Transformation

other. With the provision of dedicated methods, these POUs are able to communicate with each other in a convenient way. The usage of global communication variables is not necessary.

Section 2.2.3 on page 19 points out that, although part of the norm, the POU class has not yet been implemented in any software development environment. We thus opted for function blocks as the IEC 61131-3 counterpart for a partial-Grafcet.

3.2. Choice of IEC 61131-3 Programming Language

For the successful transformation of a hierarchical Grafcet into an IEC 61131-3 implementation, the choice of the appropriate IEC 61131-3 programming language is crucial. We define the following quality measures that have to be met to ensure practicability and acceptance of the transformation concept:

**Similarity** The most obvious point of the transformation is that the transformed code shall show the same dynamic behavior on a PLC as one would expect from the Grafcet. This point could also be called ‘equivalence’, but due to the differences in the event-based Grafcet and the cyclic-sequential PLC, similarity is a more precise expression.

**Portability** Although the IEC 61131-3 defines a global standard for the programming of PLC-systems, it still leaves room for interpretation and thus leads to vendor-dependent implementations. The transformation algorithm shall avoid such ambiguous structures to ensure that the produced code can run on any IEC 61131-3 software development environment.

**Readability** As mentioned in section 1, the specification of the control algorithm on the one hand and the implementation and maintenance of the control program on the other hand are typically done by different users with a different background. It is common practice that minor changes are done directly in the code. The produced code has to be easily readable without knowledge of the transformation algorithm. Ideally, the control flow of the automated system can be directly seen from the control program.

**Maintainability** Small changes to the control flow (e.g., different actor trigger, additional checks etc.) shall also result in only small changes in the code. Implicit relations between signals and variables should be avoided.

A transformation approach that helps to ensure some of these quality metrics has been established by Frey and picked up by Schumacher: One-to-one correspondence is an approach which implies that a Grafcet-element shall have a direct correspondence in its IEC 61131-3 implementation. The one-to-one correspondence allows an easy reinterpretation of the produced code, avoids implicit relations between variables and ensures that the user knows where to find parts of the code he needs to be changed.
3.2. Choice of IEC 61131-3 Programming Language

Furthermore, this approach is a good foundation for future improvements in the domain of model-based-development with Grafcet, e.g., bi-directional transformation (see section 6.1).

Section 2.2.2 gives a brief overview of the available IEC 61131-3 languages. This work only considers Sequential Function Charts and Structured Text as target languages for the transformation. The remaining languages were omitted due to the following reasons:

**Instruction List** The transformation of Signal Interpreted Petri Nets (SIPN) to an IEC 61131-3 implementation in [14] uses IL as a target language without further reasoning for this choice. However, the most recent version of the IEC 61131-3 norm states that as an assembler-like language, IL is outdated. It will not be included in the next version of the norm [20]. The transformation should not be based on a language that is considered ‘deprecated’ [20, p. 195].

**Ladder Diagram** Due to the fact that LD is based on the wiring of relais inside a controller, it is ‘far away’ from an abstract control-flow language like Grafcet. A LD implementation would most likely not fulfill the readability quality metric.

**Function Block Diagram** To effectively model a control-flow given in Grafcet, a FBD implementation would have to model states and transitions as function blocks and connect them. Such an implementation would look similar to an SFC implementation, so SFC could be used straightaway.

3.2.1. Analysis of Sequential Function Charts as a Target Language

Before the analysis of the potentials of SFC as a target language, this section gives a brief description of the functionality of SFC. For a detailed introduction, refer to [22] or directly to the IEC 61131-3 [20].

The structure of a SFC is based on steps and transitions, two disjoint sets of knots that are connected by directed arcs. A step consists of two properties that indicate its activity ($X$) and the time that has passed since the last activation of the step ($T$). Each step can be associated to a set of actions. Actions can either set a boolean variable or execute a procedure that can be written in any of the IEC 61131-3 languages. The dynamic execution of an action depends on its qualifier. For this work, the most important qualifiers are the S-qualifier for a stored execution of an action and the N-qualifier that activates the corresponding action for the active duration of the corresponding step. Each transition has an associated boolean transition condition. The dynamic behavior of an SFC underlies a set of evolution rules, similar to those of Grafcet and Petri nets. The initial state is marked by an initial step. The succeeding states are reached by sequential firing of transitions. A transition fires if all its preceding steps are active and its transition condition evaluates to true. Actions are executed when their corresponding step is active. Due to the cyclic nature of PLCs, these SFC execution characteristics follow a certain execution order that is repeated in each scan cycle. This major difference to the event-based Grafcet evaluation is picked up later in this section.
Most work related to the transformation of specification languages to an IEC 61131-3 implementation focuses on SFC as a target implementation language. Katzke et al. examine the transformation of UML and suggest SFC because of its capability to directly model discrete and batch processes [24]. Klein et al. analyze the transformation of a SIPN model into an SFC implementation program [26]. The authors use model checking techniques to validate the SIPN model, transform it into SFC and validate it again. They justify the choice of SFC with the syntactic similarity to SIPN, but also mention major differences that have to be considered in the context of this work as well:

- SFC does not have transient states. A state is always active for at least one PLC cycle.
- There can be only one initial step in an SFC.

These restrictions apply to Grafcet as well, because transient states and multiple initial steps are also included in Grafcet. In the validation process of the SFC using the model checker SMV, a property that has been validated on the SIPN does not hold on the generated SFC. This is due to the lack of transient state behaviour. A transformation approach that gives respect to this property would be desirable.

According to [5] and [22], the SFC language is based on the Grafcet norm IEC 60848 [18]. Because of their syntactical similarity, the two are often mixed up, especially in the anglophone application domain (e.g., [46], [13]). However, Grafcet and SFC are different in their dynamic behaviour, mostly because of the sequential character of SFC. The Grafcet norm itself explicitly mentions the important semantic differences in the appendix:

IEC 60848 and IEC 61131-3 each have a specific domain of application: a behaviour specification language (GRAFCET – GRAPhe Fonctionnel de Commande Etape Transition) independent of any specific technology of implementation, for IEC 60848, and a specific programming language (SFC – Sequential Function Chart), for IEC 61131-3. GRAFCET of IEC 60848 is used by a grafcet chart to describe/specify the behaviour of system, as viewed from 'outside' of the system, while the SFC language of IEC 61131-3 is used to describe (part of) the implemented software structure 'inside' of the system. [18, p. 52]

and further

If the two languages were both used to describe a control system, the two descriptions (two different document kinds) would in a given case look graphically similar. However, they would not have the same meaning, not even if they were graphically identical. This would just indicate that the structure of the software program, described in a software diagram, behaves in a way such that it can be described with a graphically similar grafcet chart. The properties of the underlying elements associated with the graphical element representations are nevertheless different in the two cases. [18, p. 52]
Apart from the known semantic differences between Grafcet and SFC, the lack of hierarchical elements in SFC is another issue when choosing the appropriate target language. There is yet no known effort to transform Grafcet including its hierarchical elements to SFC. David and Alla suggest to transform CIPN into SFC, but limit the scope of the interpretation algorithm to basic Grafcets without abbreviations (i.e., hierarchical elements) [11]. Schumacher includes hierarchical elements in the Grafcet specification, but merges all partial-Grafcets into one remaining Grafcet (i.e., normalization of all Grafcets) before the transformation [42]. The reason why hierarchical elements are not included is that the syntactical similarity between Grafcet and SFC is limited to elements of a basic Grafcet. There is no element in SFC that resembles the hierarchical elements macrostep, enclosure and forcing order. That means that their behavior has to be modeled with basic elements. The impact of all hierarchical elements on the dynamic behavior of a Grafcet can be traced back to the introduction of implicit links and conditions. When trying to model the hierarchical elements of Grafcet in SFC, these implicit links need to be uncovered and explicitly modeled.

We shall demonstrate on a number of examples that the implementation of such additional links can be awkward to handle and to some extent defeats the quality measure readability.

Forcing Orders in SFC

In the case of a forcing order, explicit models for the implicit links are:

1. Additional transitions from every step to the forced steps with transition condition `forcing step == 1` to bring the partial-Grafcet to the forced situation.
2. Additional condition `NOT forcing step == 1` for each existing transition so that the Grafcet does not evolve while the forcing order is active.
3. In the case of a parallel branch: A synchronization and transition from every possible combination of steps that can be active at the same time to the forced steps. This is due to the fact that a step cannot be activated twice, so two active steps that link to the same step need to be synchronized.

**Example 3.2.1.** Consider the Grafcet in figure 3.1a on the previous page. From any current situation $\text{sit}(t)$, the forced Grafcet $G1$ evolves to the situation $\text{sit}(t+1) = \{S5\}$ on activation of step $S2$ in $G\text{Main}$. An SFC that shall show similar behavior needs to have corresponding additional transitions and links. The partial-Grafcet $G1$ will stay in the situation $\text{sit}(t+x) = \{S5\}$ during the time $x$ that $S2$ is active. Even when transition $t13$ is fireable, the active forcing order suppresses its firing. All transitions in the corresponding SFC need to be extended so that they will not fire while the forcing order is active. Figure 3.1b shows such a SFC implementation that preserves the hierarchical structuring of the corresponding Grafcet and implements the implicit links.

![Grafcet Diagram](image)

**Figure 3.2.: Extended Example for the Transformation of Forcing - Grafcet**

Example 3.2.1 is still easy to read because there are only a few additional links that are easy to handle. However, if the Grafcet contains parallel branches, the number of implicit transitions increases greatly because of the necessity of combining all steps of parallel branches:

**Example 3.2.2.** The partial-Grafcet $G1$ in figure 3.2 will evolve from any situation $\text{sit}(t)$ to the situation $\text{sit}(t+1) = \{S20\}$ on activation of step $S2$ in $G\text{Main}$. Considering the shown current situation $\text{sit}(t) = \{21, 24\}$, this means that in an SFC these two steps have to be synchronized and linked to step $S20$. The necessary transition has the condition $G\text{MAIN}.S2.X$ to indicate that the forcing order is active. Such an additional synchronization needs to be implemented for all possible situations where two steps can be active at once, i.e., $\text{sit}(t) = \{21, 22\}$, $\text{sit}(t) = \{21, 24\}$, $\text{sit}(t) = \{23, 22\}$.
3.2. Choice of IEC 61131-3 Programming Language

Figure 3.3.: Extended Example for the Transformation of Forcing - SFC

and \( \text{sit}(t) = \{23, 24\} \). The SFC in figure 3.3 contains all necessary additional elements to model the Grafcet's behavior.

Comparing the original Grafcet in figure 3.2 to the corresponding SFC in figure 3.3, it becomes apparent that already quite simple Grafcets result in relatively complicated SFC implementations. Furthermore, while the given SFC is already difficult to read, the \( m:1 \) relation of forcing orders to forced Grafcets leads to additional complexity. If \( G_1 \) was forced by another forcing order in \( G_{\text{Main}} \) (i.e., a forcing order associated to step \( S_1 \)), a second set of additional transitions and transition conditions would have to be added to the chart.

The example of a forcing order underlines why the correct implementation of transient state behavior is important when modeling a Grafcet with hierarchical elements. The forcing order has the same properties as a continuous action, which means that if it links to a state that is only traversed during transient state evolution, the corresponding forced Grafcet is unaffected and continues normal evolution. In SFC however, each step is active for at least one cycle. That means that even if the forced Grafcet will not 'freeze', it will evolve to the forced situation and continue evolution from there. Before considering forcing orders, simply ignoring transient states meant that signals in continuous actions emitted a very short pulse during one cycle, which could probably be without effect on a real world plant. In our case, ignoring transient state evolution would mean severe differences in dynamic behavior.

Macrosteps in SFC

To model the implicit links of a macrostep in SFC, one needs to add:

- A link from the transition preceding the macrostep to the entry-step of the macrostep
expansion, which turns the transition into a divergent synchronization.

- A link from the exit-step of the macrostep expansion to the transition succeeding the macrostep, which turns the transition into a divergent synchronization.

Modeling these links with the macrostep and its expansion in two different function blocks requires a workaround. The function block has to be called as an action with qualifier N to make sure the function block is only active while the corresponding step is active. To model the first link, the exit- and the entry-step of the expansion have to be connected via a transition with condition 'macrostep not active'. The final action evaluation will fire that transition once the macrostep is left and bring the function block back to its desired initial state before it is deactivated. Adding an additional condition 'exit step of expansion active' to the transition succeeding the macrostep models the second link. Figure 3.4 shows such a possible SFC implementation of the Grafcet in figure 2.3 on page 9.

Figure 3.4.: Example of the Implementation of the Grafcet in Figure 2.3 in SFC

**Enclosures in SFC**

The set of implicit links of an enclosure results in the following SFC elements:

- Activation / deactivation of the enclosed Grafcet together with activation / deactivation of the enclosure.

- Additional transitions from every step to the enclosed steps with condition 'enclosure not active'.

- Additional transition conditions 'enclosure active' to ensure mutual exclusivity of concurrent transitions.

In analogy to the Macrostep model, the second link is needed to bring the function block back to the situation determined by the set of enclosed steps on deactivation of the enclosure. This way, the function block is already in the right state when the corresponding
3.2. Choice of IEC 61131-3 Programming Language

enclosure gets activated again. The SFC in figure 3.5 is an implementation of the enclosure example in figure 2.4 on page 10. While this still looks easy to handle because of a relatively small number of additional elements, Grafcets with parallel branches require the same approach that is needed in the modeling of a forced Grafcet. All possible combinations of steps that can be active at the same time need to be synchronized and then linked to the set of enclosed steps.

Further relevant characteristics of SFC

Another issue of SFC is the lack of a formal semantics, which results in vendor-dependent behavior. This has been discussed in great detail in [5], which also gives a basis for a formal semantics. The current version of the IEC 61131-3 does not include this formal semantics, so the vendor-dependent issues remain:

- The order in which simultaneously fireable transitions are fired is defined by a priority rule. However, the exact implementation of this rule is not defined, so some tools might evaluate such transitions "from left to right", others might use a different approach.

- It is not defined if transitions are evaluated and fired before actions are executed, or if the reverse order is applied. If the value of an action or transition condition is subject to the activity of a certain step in the chart (as in the case of modeling a forcing order), the evaluation order can influence the dynamic behavior.

- The priorities of a parent SFC and its nested SFCs (i.e., an action that is modeled in SFC) are not defined. Furthermore, it is vendor-dependent if a nested SFC remains in its current state when its deactivated, or if it starts with its initial step on its next activation.

The second point is also the subject of [16]. Hellgren et al. analyze the SFC execution order defined by the PLCopen committee [35], also see section 5.1.2, two European
and one Japanese system vendors and encounter three different algorithms being used. We discuss the different characteristics of possible evaluation algorithms in section 4.1.

Summary

The aim of this thesis is to find a suitable way to transform the hierarchical elements of Grafcet into an IEC implementation while preserving the hierarchical structure. This section shows that using SFC, it is in principle possible to keep the structure of a collection of partial-Grafcets and transform them separately into function blocks. A discussion of the quality metrics defined in the beginning of this chapter shows why we did not choose SFC as a target language and favored an implementation in Structured Text:

Similarity The lack of transient states is a major difference to the dynamic behavior of Grafcet. This can, for example, lead to undesired behavior of forcing orders. To overcome this, it would be necessary to limit the Grafcet design in a way that transient evolution cannot happen (e.g., mutually exclusive succeeding transitions).

Portability The different execution orders and transition evaluation priorities of the different vendors make it impossible to guarantee that the transformed code will behave the same on any hardware. This quality metrics is violated.

Readability The SFC in figure 3.3 on page 27 is a good example of how the usage of SFC violates this quality metrics. The additional elements needed to model implicit links of hierarchical element lead to confusing and hard-to-read SFCs.

Maintainability Minor modifications include the variation of transitions conditions and actions. The transformation does not add redundancy or hidden links between actions or transitions, so if the user finds the transition or actions he wants to change, it can easily be done. This quality metrics is met.

3.2.2. Analysis of Structured Text as Target Language

Structured Text programs consist of a number of statements that are evaluated from top-to-bottom during program execution. Statements include variable assignments, different kinds of selections (IF, CASE) and different kinds of loops (FOR, WHILE, REPEAT). Statements can also consist of function block or method calls. These are all principles well known from languages like C and PASCAL. Again, for further reference see [22] or directly the IEC 61131-3 [20].

Although [14] uses Instruction List for transforming signal interpreted Petri nets to IEC 61131-3, which we did not consider for our work, the general approach is very promising. Places, which correspond to steps in Grafcet, are modeled as boolean variables that indicate whether a place is marked or unmarked. The program constantly evaluates all transitions as to if they are enabled and fireable. If so, the boolean variable for the preceding step is deactivated and the succeeding step activated. This textual implementation of a sequencer is a quite basic approach in PLC programming. Using
3.2. Choice of IEC 61131-3 Programming Language

the same concept, an implementation of a sequencer in Ladder Diagram is possible \[16\]. In essence, such an implementation is a textual description of the Grafcet’s dynamic behavior and enables the explicit modeling of the systems execution, which is implicit in SFC and cannot be altered.

Such an implementation in Structured Text requires the definition of a Grafcet interpretation algorithm for sequential systems. The development of this algorithm is part of this work in section 4.1. With the object-oriented features of function blocks introduced in section 2.2.3, each function block that represents a partial-Grafcet provides a standard set of methods that correspond to the steps of such an algorithm. Dedicated methods to alter a function block’s state from the outside simplify the modeling of implicit links of hierarchical structures severely. Using this approach, an additional POU would be needed that instantiates all function blocks and implements the interpretation algorithm via calling the specific methods of each function block.

The author of \[14\] suggests three different ways to take care of transient state evolution. In a direct modeling approach, a boolean variable $\delta$ is defined that is set to false before the evaluation of transitions. If a transition fires, $\delta$ is set to true. This cycle executes until $\delta$ remains false during the transition evaluation, which indicates that a stable state has been reached. Two alternative concepts deal with the fact that the ordering of the transition code segments influences the code execution. The pseudo-code segment in listing 3.1 illustrates this behavior.

```
1  //Transition 0
2  IF transition0 enabled AND Place0 marked THEN
3  unmark Place0
4  mark Place1
5  END IF
6
7  //Transition 1
8  IF transition1 enabled AND Place1 marked THEN
9  unmark Place1
10 mark Place2
11 END IF
```

Listing 3.1: Pseudo-code for state evolution after \[14\]

Assuming that transition0 and transition1 are true and Place0 marked, we would see transient behavior in the given order. After code execution, Place2 is marked and Place1 seems to be skipped. However, when we change the evaluation order, we evaluate transition 1 first. This evaluates to false because Place1 is not marked. Afterwards, we evaluate transition 0 which unmarks Place0 and marks Place1. In this case, after code execution, we see that Place1 is marked. Using a simulative respectively analytical determination approach, a code segment ordering that ensures transient behavior can be computed. In this work, we shall use the direct modeling approach to keep the program simple.

Textual source code written in Structured Text or Instruction List can be compiled and executed on any type of PLC. The execution does not depend on any vendor-specific interpretations.
The discussion of the quality metrics shows why we favored Structured Text as a target language for the transformation. Some of the discussed advantages are introduced in section 4.2.

**Similarity** The possibility of implementing transient behavior is a big step towards behavioral similarity between Grafcet and the PLC implementation. Furthermore, we can directly implement a Grafcet interpretation algorithm for sequential systems and influence the evaluation order of elements.

**Portability** A Structured Text control program can run on any type of PLC. There are still some limitations regarding the portability, but these result mostly from different implementation stages of new IEC 61131-3 elements and data exchange formats (see section 5.1.2).

**Readability** Naturally, a control program written in a textual language requires more work to understand the program than a simple graphical representation. However, the implicit links and additional elements needed to model hierarchical behavior are separated from the standard set of methods. The initial Grafcet can easily be derived analyzing only a subset of the Structured Text program, which we discuss in section 6.1.1 on page 59.

**Maintainability** To change a transition condition or an action’s output behavior, only one line in one method needs to be changed. This quality metrics is met.
4. Transformation Concept

With a concept for a program structure and the choice for a target language for the transformation approach, the next step is to find a representation of Grafcet elements in Structured Text to be able to derive an IEC 61131-3 program with equivalent dynamic behavior. Section 4.1 is a discussion of how to interpret the evolution rules of Grafcet on a sequential machine. The succeeding section contains the actual transformation concept and shows how any Grafcet can be transformed into an IEC 61131-3 representation using this concept.

4.1. Grafcet Interpretation Algorithm

The attempt to find a formal definition of Grafcet in [42] results in an interpretation as a control interpreted Petri net. This approach is based on [11], where a discussion of the similarity of Grafcet and CIPN together with the *Algebra of Events* results in the two following statements:

- If a control interpreted PN is interpreted as a Grafcet, it has exactly the same behavior.
- If a Grafcet is interpreted as a control interpreted PN, it has exactly the same behavior. [11, p. 357]

The authors give an interpretation algorithm for a CIPN and explicitly state that this algorithm could be a basis for transforming Grafcet into an IEC 61131-3 implementation. The algorithm gives respect to transient evolution and includes an evolution cycle that tries to find a stable marking.

However, they also underline that the CIPN interpretation algorithm is limited to a basic Grafcet, i.e., a Grafcet without the hierarchical elements. We use this interpretation algorithm and extend it by the notion of the hierarchical elements *macrostep*, *enclosure* and *forcing order*. The resulting algorithm (listing 1 on page 35) is the basis for the implementation of Grafcet in an IEC 61131-3 code environment.

Experimenting with different possible implementations for the execution algorithm, we came across some interesting properties related to the execution order of the different steps. It turns out that these properties apply to all execution models of step / transition systems and are discussed in greater detail in [16] on the example of SFCs. There are two main aspects of the execution of Grafcet. One is the evolution model, i.e., the firing of transitions and activation / deactivation of corresponding steps. The other is the activity model, i.e., the execution of actions. Concerning the evolution model, [16] differentiate between the *deferred transit evolution model (DTE)* and the *immediate transit evolution model (ITE)*. In the DTE, all transition conditions are evaluated first, afterwards all transitions that were marked fireable fire one by one. The ITE evaluates a transition condition and immediately fires a transitions if it is fireable before it continues with the evaluation of the next transition. To model the behavior of Grafcet that all
transitions that are fireable fire at the same time, we need to use the DTE. Otherwise, the firing of a transition $t_1$ could remove the token from a predecessor step of another transition $t_2$. Thus, this transition would not be able to fire anymore, although it was supposed to fire together with $t_1$. Unlike [16], we also consider transient evolution for our algorithm. During transient evolution, the transitions only "see" that step activity variables change, while all other variables remain unchanged. This is due to the fact that such an evolution has theoretically no duration in Grafcet. To achieve transient evolution, we need to distinguish between the set of enabled transitions $\text{EnTrans} = \{ t \in T_G \mid t.ECond == \text{true} \}$ and fireable transitions $\text{FireTrans} = \{ t \in \text{EnTrans} \land \forall s \in t.S_B \mid s.x == \text{true} \}$. Transitions are marked as enabled before an evolution cycle. During the evolution cycle, only the fireability of transitions is updated.

The activity models can be distinguished in a deferred action activity model (DAA) and an immediate action activity model (IAA). In correspondence to the evolution models, the actions in the DAA are executed after all transitions have fired and all steps are activated / deactivated. In the IAA, the execution of an action is directly linked to the activation / deactivation of a step. Since continuous actions in Grafcet are only executed once a stable marking has been reached, using the IAA for continuous actions, however, is not possible. In the evolution cycle, a stable marking has to be found first. On that basis, the continuous actions can be executed one by one afterwards. For stored actions, both approaches are possible when using the DTE model. For the sake of simplicity, we decided to use the DAA model for both stored and continuous actions.

Note that these models and the resulting algorithms only consider the order of the different execution steps of a Grafcet interpretation. They do not include any information on the order in which the several sets of elements are evaluated (e.g., the order in which transitions are checked for fireability). We further discuss this order in section 4.2.

On a PLC implementation, the algorithm step 13 marks the end of a scan cycle. At this point, the outputs are written into the output register and the input values of all signals are read and updated.

We added the steps 5, 6, 9 and 11 to the existing algorithm and modified the steps 3 and 8 so that the notion of events is not necessary in the algorithm anymore, because the execution on a PLC is triggered by the scan cycle. An assumption underlying the handling of macrostep expansions is that the transition preceding a macrostep can only fire when the exit-step of the expansion is active. This can be achieved by including the exit-step of the expansion to the set of preceding steps of the transition. Extending the transition condition with a respective statement is not an option, since this only affects the enabling, but not the fireability of the transition. Though, this means that a macrostep expansion cannot be traversed entirely during transient evolution. The same behavior without this assumption can also be achieved via manipulating the transition condition on activation of the entry step and exit step, similar to the forcing activities. This has in fact been done in the transformation prototype.

The presented algorithm requires some restrictions on the design of the Grafcet. These restrictions result from the difference of event-based Grafcet theory to sequential-cyclic real-world PLC programming.
Algorithm 1 Grafcet Interpretation Algorithm

1: Initialization: Activate all initial steps, execute the corresponding stored actions.
   for all (s ∈ $S_{InitG}$)
   
   \[ s.x = true \]
   
   end for
   
   for all (a ∈ $A_{GS}$ | a.inst == StepActivation ∧ inst(a.s.x) == a.inst)
   
   \[ a.o = a.val \]
   
   end for

2: Compute the set of enabled transitions, i.e., all transitions where the transition condition evaluates to \textit{true}:
   \[ EnTrans = \{t \in T_G \mid t.ECond == true\} \]

3: Compute the set of fireable transitions, i.e., all enabled transitions where all predecessor steps are active:
   \[ FireTrans = \{t \in EnTrans \land \forall s \in t.S_B \mid s.x == true\} \]

4: Fire all fireable transitions:
   for all (t ∈ FireTrans)
   
   for all (s ∈ t.S_B)
   
   \[ s.x = false \]
   
   end for
   
   for all (s ∈ t.S_A)
   
   \[ s.x = true \]
   
   end for
   
   end for

5: Activate the entry-step of all macrostep expansions that have been inactive before and where the corresponding macrostep is active now. Deactivate the exit step of all macrostep expansions where the corresponding macrostep is unactive now.
   for all (c ∈ $C_M$)
   
   if (c.m.x == true ∧ \forall s ∈ S(cm) | s.x == false)
   
   \[ c.s_I = true \]
   
   end if
   
   if (c.m.x == false ∧ c.s_O == true)
   
   \[ c.s_O = false \]
   
   end if
   
   end for

6: Activate the enclosed steps of all enclosed partial-Grafcs that have been unactive before and where the enclosure is active now. Deactivate all steps of all enclosed partial-Grafce where the corresponding enclosure is unactive.
   for all (c ∈ $C_E$)
   
   if (c.xm.x == true ∧ \forall s ∈ S(enc) | s.x == false)
   
   for all (sx ∈ s.S_enc)
   
   \[ sx.x = true \]
   
   end for
   
   end if
   
   if (c.xm.x == false)
4. Transformation Concept

for all \( s \in S(\text{enc}) \)
\[ s.x = \text{false} \]
end for
end if
end for

7: Execute all stored actions associated with the steps that were marked active / inactive with the previous step according to their allocation instant:
for all \( (a \in A_{GS} \mid a.\text{inst} == \text{inst}(a.s.x)) \)
\[ a.o = a.val \]
end for

8: Determine the set \( S_N \subseteq S_G \) of steps that have changed their active value during the execution of steps 4, 5 and 6. If \( S_N \) is not empty \( (S_N \neq \emptyset) \), return to step 3. Otherwise, a stable marking has been reached.

9: Deactivate all forcing orders that are connected to inactive steps: Remove the 'not enabled' marking from all transitions of the forced Grafcet.
for all \( (a \in A_{GF} \mid a.s.x == \text{false}) \)
for all \( (t \in a.c.T) \)
Remove \text{false} from \( t.E_{Cond} \) and evaluate.
end for
end for

10: Deactivate all continuous actions that are connected to inactive steps.
for all \( (a \in A_{GC} \mid a.s.x == \text{false}) \)
\[ a.o = \text{false} \]
end for

11: Execute all forcing orders that are connected to steps that are part of the stable situation: Activate the forced situation, mark all transitions as not enabled.
for all \( (a \in A_{GF} \mid a.s.x == \text{true}) \)
for all \( (t \in a.c.T) \)
Add \text{false} to \( t.E_{Cond} \) and evaluate
end for
if \( (a.op == \text{Current}) \)
for all \( (s \in a.c.S) \)
\[ s.x = \text{false} \]
end for
end if
if \( (a.op == \text{Init}) \)
for all \( (s \in a.c.S \mid s \in S_{\text{InitG}}) \)
\[ s.x = \text{true} \]
end for
end if
if \( (a.op == \text{Spec}) \)
for all \( (s \in a.c.S \mid s \in a.sit) \)
\[ s.x = \text{true} \]
end for
end if
end for
12: Execute all continuous actions that are connected to steps that are part of the stable situation.

for all \((a \in A_{GC} \mid a.s.x == \text{false})\)

\(a.o = a.E_{Cond}\)

end for

13: Return to step 2.

The most important aspect is the necessity of mutually exclusive concurrent transitions. Consider figure 4.1a. The Grafcet norm states that ‘Exclusive activation of a selected sequence is not guaranteed from the structure. The designer should ensure that the timing, logical or mechanical aspects of the transition-conditions are mutually exclusive.’[18, p. 33]. For the given figure, this means that when step 20 is active and signal \(a\) becomes 1, both transition \(t_1\) and \(t_2\) fire and the steps 21 and 22 become active. The interpretation algorithm would yield the same result. According to the above quote, this is in line with the norm, although not desired. If the designer intends such a ‘token split’, he shall use a synchronization as in figure 4.1d. In the case of figure 4.1b, one has to distinguish between the Grafcet theory and the way the interpretation algorithm works. According to the Algebra of Events, two independent signals can never occur at the same time. For the example, this means that if signal \(a\) and \(b\) are independent, \(t_1\) and \(t_2\) will never fire simultaneously and the behavior is well-defined. Given the implementation of the interpretation algorithm on a PLC, however, it could happen that the signals \(a\) and \(b\) change their value not at the same time, but during the same scan cycle. During step 2 and 3 of the algorithm, both transitions \(t_1\) and \(t_2\) would be marked as fireable and then fired in step 4, resulting in the situation where steps 21 and 22 are active. To avoid such behavior, the following requirement has to be met:

**Definition 4.1.1.** The transition conditions of concurrent transitions have to be designed mutually exclusive to avoid unexpected token splits. (figure 4.1c)

Another important remark on the design of a Grafcet concerns conflicts in the allocation of signals. In figure 4.2 on the next page, the firing of transition \(t_1\) leads to the activation of steps 30 and 31 and thus a conflict in the value of \(k\). The norm has no rule of how to handle such conflicts and states that ‘the designer shall ensure that two contradictory allocations cannot occur simultaneously’[18, p. 19]. In the interpretation as a CIPN, this case is covered as well. A CIPN has to be safe to be interpreted as a Grafcet.
One property of a safe CIPN is that for every pair of operations that can be active at the same time, these operations need to be strongly compatible. This means that the sets of variables manipulated by operation $O_I$ needs to be disconnected from the sets of variables manipulated by operation $O_J$. Our interpretation algorithm however would not recognize such a conflict. All actions are executed one by one after one evolution cycle. The output value of $k$ would be determined by the last action that manipulated that value and thus depend on the execution order. In the case of our implementation (top-to-bottom, left-to-right execution), $k$ would be 10. This can be desired, but the behavior of a control program should be explicit, so that the following design restriction applies:

**Definition 4.1.2.** For every pair of stored actions that can be active at the same time, the sets of variables they manipulate need to be disjoint.

For continuous actions, such conflicts are not possible. The outputs of continuous actions are `true` if a corresponding action is active and `false` otherwise. If several continuous actions try to manipulate the same output, it simply is `true`.

Especially in the case of hierarchically structured Grafcets, the design restriction regarding stored actions seems very difficult to achieve. Theoretically, the designer has to check every possible system state, i.e., every possible situation over all partial-Grafcets, to ensure that no conflicts occur. ‘State explosion’ with extension of the program by more and more partial-Grafcets comes to mind. Yet, in a real world control program, different hierarchically structured partial-Grafcets have different control tasks and thus usually operate on a different set of signals and variables. This facilitates the handling of the given restriction. In section 6.1, we give a brief outline on how static code analysis can help in meeting the design restrictions.

### 4.2. Resulting Transformation Concept

Following the approach of implementing the Grafcet interpretation algorithm presented in section 3.2.2, we need a boolean variable for every step and transition. To model the dynamic behavior according to the Grafcet interpretation algorithm for sequential systems, a standard set of methods represents each step of the algorithm. Table 4.1 shows a mapping of these methods to the algorithm steps with a short comment on the functionality.
4.2. Resulting Transformation Concept

<table>
<thead>
<tr>
<th>Step</th>
<th>Corresponding method and comment</th>
</tr>
</thead>
</table>
| 1    | **IF startProgram THEN**  
The initialization on startup is only done by the main program. Initial states are already set to 1 by their default value in the function block’s interface. On start of the system, the program checks a variable `startProgram` with default value `true` and sets it to false, followed by calls of all `storedActions()` methods of all function blocks. This part of the program is thus only executed once. |
| 2    | `evaluateTransitions()`  
Evaluates the transition conditions and assigns the result to the boolean variable of the corresponding transition, e.g.:  

```
t1 := a AND NOT b;
``` |
| 3    | `evaluateFireableTransitions()`  
Evaluates for each transition if the transition condition is met and if all predecessor steps are active. Writes the result into a second boolean variable of the corresponding transition, e.g.:  

```
ft1 := t1 AND S3 AND S4;
``` |
| 4    | `stateEvolution(stableState: BOOL) returns BOOL`  
Checks for each transition if its fireable. Deactivates the predecessor and activates the successor states. Sets the `stableState` flag to false to indicate that a change has happened (necessary for transient state evolution)  

```
IF ft1 THEN  
    S3 := 0;  
    S4 := 0;  
    S5 := 1;  
    stableState := 0;  
END_IF;
``` |
| 5, 6 | integrated in `storedActions()`  
Checks for each macrostep / enclosure if it is active. Sets the `active` flag or calls the stopping function of the corresponding function block respectively.  

```
IF M1 THEN  
    fb_M1.activate();  
ELSE  
    fb_M1.initializeAndStop();  
END_IF;
``` |
### 4. Transformation Concept

<table>
<thead>
<tr>
<th>Step</th>
<th>Corresponding method and comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td><strong>storedActions()</strong>&lt;br&gt;Checks for each step if its active and executes the corresponding stored action, e.g.:&lt;br&gt;<code>IF S5 THEN&lt;br&gt;   output1 := 5;&lt;br&gt;END_IF;</code></td>
</tr>
<tr>
<td>8</td>
<td><strong>WHILE stableState := FALSE DO</strong>&lt;br&gt;The main program calls all <code>stateEvolution</code> functions of all function blocks repeatedly in a while-loop. The <code>stableState</code> flag variable is the input and output parameter. The exit condition of the while-loop is that <code>stableState</code> remains unchanged during one loop.</td>
</tr>
<tr>
<td>9</td>
<td>integrated in <strong>pullDownSignals()</strong>&lt;br&gt;Removes the <code>forcedFlag</code> from forced Grafcets when the corresponding step is not active anymore:&lt;br&gt;<code>IF NOT S5 THEN&lt;br&gt;   fb_G1.setForcedFlag(FALSE);&lt;br&gt;END_IF;</code></td>
</tr>
<tr>
<td>10</td>
<td><strong>pullDownSignals()</strong>&lt;br&gt;Sets all outputs of continuous actions to 0, e.g.: <code>output10 := 0;</code></td>
</tr>
<tr>
<td>11</td>
<td>integrated in <strong>continuousActions()</strong>&lt;br&gt;Carries out all necessary actions to move a forced partial-Grafcet to the forced situation and sets the <code>forcedFlag</code> to true to keep it there. For example, the forcing of a specific situation yields:&lt;br&gt;<code>IF S2 THEN&lt;br&gt;   fb_G1.deactivateAllSteps();&lt;br&gt;   fb_G1.setForcedFlag(TRUE);&lt;br&gt;   fb_G1.activateStepS7();&lt;br&gt;   fb_G1.activateStepS8();&lt;br&gt;END_IF;</code></td>
</tr>
<tr>
<td>12</td>
<td><strong>continuousActions()</strong>&lt;br&gt;Checks for each state if its active and sets the output of the corresponding continuous action to <code>true</code>.&lt;br&gt;<code>IF S5 THEN&lt;br&gt;   output10 := true;&lt;br&gt;END_IF;</code></td>
</tr>
</tbody>
</table>
4.2. Resulting Transformation Concept

Step | Corresponding method and comment
--- | ---
13 | **EOF program**

The end of a program marks the end of a scan cycle.

| Table 4.1.: List of methods corresponding to the steps of the Grafcet interpretation algorithm in section 4.1 |

This implementation includes some variations of the initial Grafcet interpretation algorithm. For performance reasons, we changed the way how macrostep expansions and enclosed partial-Grafcets are activated/deactivated (steps 5,6). In the algorithm, the entry-step / enclosed steps are activated on activation of their parent steps. Accordingly, all steps are deactivated on deactivation of their parent steps. Given the structure of the main program that calls all methods, this would mean that each if-statement of each method would be evaluated, although the function block is not active. The results would be correct since all if-statements would detect inactive steps, but the performance of the program might suffer. We improved this by introducing an active flag for macrostep expansions and enclosed partial-Grafcets, which is set / unset together with the parent element. The methods for state evolution and action execution are only executed under the condition that the active flag is set.

The same principle applies to the execution of fording orders. Once a forcing order is active, it sets a forcedFlag within the forced function block. An active forcedFlag prevents the execution of the state evolution function. Setting all transitions to ‘not fireable’ would yield the same behavior but require more computations.

All output variables of continuous actions are set to 0 in each cycle and then reset to 1 if a corresponding continuous action is executed. This is to ensure correct behavior when macrosteps are traversed in a single transient evolution step. This variation does not affect the system behavior because the state of the variables is intermediate during a scan cycle and only written to the actual output at the end of such a cycle.

Furthermore, we merged the activation and deactivation of macrosteps and enclosures into the method for stored actions and the forcing orders into the method for continuous actions. This decision is rather intuitive because it simplifies the program structure.

The necessity of additional methods to model implicit links depends on the type of partial-Grafcet. For the different types, these additional methods are:

**Classical chart.** A classical chart can always be a forced Grafcet and thus needs to provide corresponding methods:

- **activateStateS*():** Necessary to activate steps of a forced situation. Since a forced Grafcet does not know about its possible forced situations, this method has to be provided for every step.
- **activateInitSituation():** Necessary to activate the initial situation (not known from the outside)
4. Transformation Concept

- **deactivateAllSteps():** First step in bringing the function block to the forced situation.
- **get/set forcedflag:** Getter and setter for the forcedflag that prevents transitions from firing.

**Macrostep expansion:**

- **activate():** Sets the active flag to true.
- **initializeAndStop():** Activates the entry step and sets the active flag to false so that no transitions fire and no actions execute anymore.
- **getFinishedFlag():** Returns true if the exit step of the macrostep expansion is active. This function has to be part of the fireability condition that succeeds the macrostep.

**Enclosed partial-Grafcet**

- **activate():** Sets the active flag to true.
- **initializeAndStop():** Activates all enclosed steps and sets the active flag to false so that no transitions fire and no actions execute anymore.

This method-oriented approach requires a coordinator that repeatedly calls the methods of the several function blocks, i.e., a Structured Text representation of the Grafcet interpretation algorithm. We shall refer to this program as the main program in the context of this work. Listing 4.9 shows a main program that coordinates one simple partial-Grafcet G1.

```structuredtext
IF startProgram THEN
    startProgram := FALSE;
    fb_G1.storedActions();
END_IF;

fb_G1.evaluateTransitions();

globStableState := FALSE;
WHILE globStableState = FALSE DO
   .globStableState := TRUE;
    fb_G1.evaluateFireableTransitions();
    globStableState := fb_G1.stateEvolution(stableState := globStableState);
    fb_G1.storedActions();
END_WHILE;
fb_G1.pullDownSignals();
fb_G1.continuousActions();
```

Listing 4.9: Example of the Main Program in Structured Text

On the implementation level, the main program is the POU that is linked to the PLC task manager.

An important aspect of the transformation concept is the evaluation order of methods and elements. Since the PLC works in a sequential way and executes the program line-by-line, the evaluation of Grafcet elements depends on their position in the source
4.2. Resulting Transformation Concept

When the design restrictions presented in section 4.1 are met, the evaluation order does not have an effect on the dynamic behavior of the Grafcet. Nevertheless, we have to define an execution order that follows a defined rule to ensure transparent program behavior. An obvious approach would be to evaluate the elements according to their position on the chart, e.g., from top to bottom and from left to right. For the example in figure 4.1b and the step `evaluateTransitions`, this means that the line that evaluates the transition condition of \( t_1 \) is put above the line for \( t_2 \). The Grafcet syntax itself does not include information about the position of elements. This information needs to be directly derived from the editor. The activation of steps of a forced situation depends on the ordering of steps in the forcing order, which is evaluated from left to right. The execution order is also important for the main program. The order in which the methods of the different function blocks are called has to follow a certain principle. We simply used the internal ordering of charts in the Grafcet editor, since the syntax does not include any priority measure as well. We pick up this aspect in section 6.1 where we discuss how the ordering can be improved.

4.2.1. Transformation Procedure

As described in the introduction, the general objective of the transformation is to create a separate POU, i.e., a function block, for each partial-Grafcet. Irrespective of the type of Grafcet chart, each function block consists of the set of basic methods in table 4.1. In addition to that, each function block has to provide the additional set of methods depending on its type. We refer to a function block that includes a set of methods with empty bodies as a `shell`. In appendix A.1.2 we show the shells for the different types of partial-Grafcets.

Our aim is to develop a one-to-one correspondence transformation approach. We limit this one-to-one correspondence to a unidirectional transformation from Grafcet to IEC 61131-3 for this work, although the final concept can also be applied to a bidirectional...
approach (see section 6.1.1). If we pick any Grafcet element, e.g., a transition, there is a corresponding set of Structured Text elements that we include in the control program. After each Grafcet element has been read and the corresponding Structured Text elements have been added to the control program, the transformation is done and the resulting program can be executed and behaves according to the underlying Grafcet. Figure 4.3 on the preceding page shows the main steps in the transformation process. In [42], the approach is the same, only that SFC is the target language. If the transformation finds a Grafcet transition, it creates a SFC transition with the same condition. In contrast to that, our transformation approach requires that several Structured Text constructs are added to different parts of the control program. For the example of a transition that means: If the transformation algorithm finds a Grafcet transition $t_1$ with the condition $\text{cond}$, predecessor state $S_1$ and successor state $S_2$ in a partial-Grafcet, it

- creates the boolean variables $t_1$ for the evaluation of the transition condition and $ft_1$ for the evaluation of the fireability of the transition. It adds these to the interface of the function block.
- adds the line $t_1 := \text{cond}$; to the method $\text{evaluateTransitions()}$ of the function block.
- adds the line $ft_1 := t_1 \text{ AND } S_1$; to the method $\text{evaluateFireableTransitions()}$ of the function block.
- adds the block below to the method $\text{stateEvolution()}$ of the function block.

```plaintext
IF $ft_1$ THEN
  stableState := false;
  $S_1$ := 0;
  $S_2$ := 1;
END_IF;
```

For each Grafcet element, we define these transformation instructions or transformation rules in section A.1
5. Prototypical Implementation of Transformation Concept

The work on this thesis includes the development of a prototypical transformation program that shall serve as a proof-of-concept for the preceding considerations. Section 5.1 gives an overview about the tool landscape we choose to embed the transformation program. The actual implementation is explained in section 5.2. The basis of the transformation program are the code shells for function blocks and the transformation rules in section A.1. This chapter closes with a discussion about the transformation program’s capabilities (section 5.4). This includes the description of test scenarios we chose to test both the transformation program and the execution algorithm for correctness. We also discuss the features of Grafcet that have not yet been added to the program and that are part of future implementations.

Figure 5.1 is an extension to the process diagram in figure 4.3 and shows the different tools and file formats used during the transformation process.

5.1. Tool Support

Part of the work of Schumacher in [42] was the development of a Grafcet-editor based on Microsoft Visio (MS Visio) and a prototype for a transformation program written in C#. Both programs serve as a basis for the transformation program that has been developed during the work on this thesis. This section gives a brief overview of their functionality. For a more detailed explanation, refer to [42] and [41].

5.1.1. IEC 61131-3 Software Development Environment

In the field of automation software development tools, well-established programming tools include the following: STEP7 by Siemens [44], Codesys by 3S-Software [1], Multi-prog by Phoenix Contact Software [33] or Control Build by Dassault Systèmes [9]. The IEC 61131-3 is a standard that harmonizes the design of industrial control programs. As such, a transformation program that chooses one of the defined programming languages as a target for the implementation should be able to produce code that runs on any
IEC 61131-3 software development environment. However, the tools differ slightly from another. None of the above tools offers full support for the latest version of the IEC 61131-3. Codesys 3.5 offers an implementation of object-oriented extensions for function blocks, but does not include the new POU-type Class. Furthermore, these object-oriented extensions have been in place since 2007 [47], six years before the corresponding norm introduced these features. That explains why the syntax of object-oriented function blocks in Codesys differs slightly from the norm, as we will explain in the next section. The current releases of Multiprog and Control Build do not support any of the new features at all. On that basis, we chose Codesys as a target software development environment. The next section explains how the choice of an input interface for the transformed control program leads to further need for tailoring the transformation program to Codesys.

5.1.2. PLCopenXML

To integrate the automatically generated ST-code into the IEC 61131-3 software development environment, we need a comprehensive and preferably standardized and vendor-independent interface. PLCopenXML is a data exchange format provided by the PLCopen organization which meets these requirements. PLCopen is a vendor- and product-independent worldwide association, whose prime mission is to develop and promote the use of international standards in the field of industrial automation [36]. A core activity of PLCopen is the certification of IEC 61131-3 compliant programming systems. However, the certification itself includes several levels (Base Level, Conformity Level, Reusability Level), that guarantee conformance with the norm to a certain degree, but still leave space for vendor- and tool-specific implementations. As the PLC-manufacturers are keen on keeping a unique selling proposition and thus a competitive advantage in the market, these vendor-specific modifications are common practice. As such, it is not possible to simply use a set of standardized project files that can be used and compiled by several different software development tools. To be able to exchange programs, libraries, and projects regardless of such often minor implementation differences, the PLCopen organization decided on XML (eXtended Markup Language) as the appropriate data language and founded a workgroup named TC6 for XML. The result of the workgroup’s effort was a first version of an XML schema published in 2005 as version 1.01. The current version has the number 2.0 and was published in 2008 [37]. More recent efforts on data exchange formats in the automation domain include the work on AutomationML, an initiative started in 2006 by an industry consortium and research institutes [2]. AutomationML does not intend to be a completely new data format, but to combine existing data formats into one convenient tool. These existing tools include CAEX [21] as a top-level format for storing the plant topology and communication information, COLLADA [25] for geometry and kinematics and PLCopenXML for behavior and sequencing descriptions. AutomationML has successfully been certified as the international standard IEC 62714-2 "Engineering data exchange format for use in industrial automation systems engineering - Automation markup language - Part 2: Role class libraries" in 2015 [17]. The very recent certification date together with the overall aim
of AutomationML and the industry impact of the involved parties (i.e., Daimler, ABB, KUKA, Rockwell, Siemens among others) allow for the conclusion that AutomationML has the potential to become the data exchange format of choice for future automation software development tools. However, all known engineering tools do not yet support the usage of AutomationML with dedicated import / export functionality. We thus focused on PLCopenXML as a target format for the transformation algorithm.

The PLCopenXML schema maps the structure of an IEC 61131-3 program or POU into a corresponding XML-tree. Figure 5.2 shows the structure of a POU in PLCopenXML. It consists of a set of attributes like name and type of POU, an interface that contains the declaration of all variables and a body. If SFC is used, a POU has an additional set of actions and transitions. This structure is following the current standard of software development tools rather than strictly following the norm. For the declaration of simple function blocks in the ST language, the IEC 61131-3 suggests one coherent code block that includes a variable declaration block in the beginning, followed by the function block’s body. The norm does not explicitly mention a split of a POU into an interface and a body. Yet, in all known IEC 61131-3 software development tools, the declaration part and the body are two different entities in two different editor windows.

The current version of PLCopenXML is based on the second edition of the IEC 61131-3 from 2003, so there is no support for the new object-oriented features that were introduced in the latest edition. As we mentioned before, Codesys 3.5 does provide the object-oriented extensions for function-blocks. However, the structure of the implementation differs from the norm, which claims:

![Figure 5.2: PLCopenXML Representation of an IEC 61131-3 POU](image-url)
The declaration of a method shall comply with the following rules additionally to the rules concerning methods of a class:

1. The methods are declared within the scope of a function block type.

2. In the textual declaration the methods are listed between the function block declaration part and the function block body. [20, p. 149]

Because Codesys handles the declaration part and the body as two different entities, the exact implementation of rule 2 would not be possible anyway. In fact, in Codesys each method is a separate entity that is mounted under the corresponding POU. To include methods into a PLCopenXML document, it is thus not possible to add the method definitions to the top of a function-block’s body, which is what the norm suggests. Codesys would not be able to interpret the code. Figure 5.3 shows a comparison of a simple function-block that includes a method in Codesys and according to the IEC 61131-3. To find out if and how the declaration of methods is supported with Codesys’ PLCopenXML import and export capabilities, we created a test program that contained all elements that are necessary for the Grafcet transformation and exported them to a PLCopenXML file. It can be seen from that file that Codesys uses an addData element for such object-oriented features. To increase flexibility of the PLCopenXML standard and to facilitate the PLC vendors to integrate vendor-specific data into a program, each functional element contains such an additional element addData. In the case of object-oriented extensions for function blocks, all methods that belong to a function block are wrapped into an addData element on the level below the POU-element. This element is highlighted in the XML-tree in figure 5.2.

![XML-tree comparison image]

Figure 5.3.: Comparison of Function-Block Implementation in Codesys and According to IEC 61131-3

Because our transformation program shall serve as a proof-of-concept for the transformation rules and shall as such be able to produce usable and testable programs, we followed the structure given by the Codesys implementation instead of sticking to the definitions of the IEC 61131-3.
5.1. Tool Support

5.1.3. Microsoft Visio Grafcet Editor

All known Grafcet editors use proprietary file formats to store the information about the designed Grafcet charts. A new editor had to be designed that stores the Grafcet information in an open file format. MS Visio is a tool for the design of graphs, networks and maps that offers the implementation of custom shape libraries and allows extensive modifications via its Visual Basic for Applications (VBA) module. Furthermore, it is possible to save a MS Visio document in the XML file format .vdx (Visio XML Format). With a custom shape library that covers all elements defined in the Grafcet specification (refer to section 2.1 on page 5), MS Visio can be used to design Grafcet charts. The current implementation of the shape library has a few restrictions:

- The forcing of a partial Grafcet allows the forcing of a maximum of three steps.
- An enclosure can only enclose one partial Grafcet, thus leading to a 1:1 relation of enclosures to enclosed partial-Grafcets instead of 1:n.

In this work, MS Visio with the integrated Grafcet shape library is referred to as Grafcet editor. The editor strictly follows the naming convention of steps and partial Grafcets according to the norm [18]. This is particularly important for our transformation program, which uses these names to distinguish between the different types of partial-Grafcets. A partial-Grafcet with the name X2/* is an enclosed partial-Grafcet that belongs to the enclosing step 2. A partial-Grafcet with the name M* is a macrostep expansion of the macrostep with name *. If neither of these properties is met, the partial-Grafcet is a classical chart. A consistency check warns the user if elements are not properly connected and if all connected Grafcets actually exist. This consistency check is purely syntactical. It is possible to design and export erroneous and/or incomplete Grafcets. The user has to make sure that the Grafcet is correct. A static analysis of Grafcet could be done using the exported Grafcet information and is part of ongoing research (see section 6.1).

5.1.4. Read-Module of Transformation Program

The decision for C# as an implementation language for the transformation program results mainly from the fact that C# is used in the author’s institute to teach object-oriented programming to students. Thus all involved programmers had a common ground in C# programming experience. With Visual Studio [29] and .NET [28], Microsoft provides a powerful framework for the development of C# applications. The structure of the transformation program is based on a Grafcet-based internal object-model. The class XMLReader takes the Visio XML file as a stream, identifies the type of element and creates the corresponding object in the internal object model. The root element document of this object model contains a container (i.e., a List in this case) of page objects, where each page represents a partial-Grafcet. All information about Grafcet-elements and their links is stored in specific lists and objects inside such a page-object. Figure 5.4 on the next page shows a simplified UML diagram of the internal object model. The transformation process consists of three main steps, represented by
5. Prototypical Implementation of Transformation Concept

Figure 5.4.: Simplified UML-Diagram of Internal Object Model

three classes. The XMLReader class reads the Visio XML file according to the above description. The class XMLConverter connects all objects using the unique VisioID of each element. Furthermore, it carries out the normalization of hierarchical elements into one single Grafcet. The last task is the preparation of all objects so that they can directly be transferred into PLCopenXML elements without further conversion during the transformation. The class PLCopenXMLWriter then takes all relevant lists of state, transition and action objects and creates the corresponding XML-elements.

To implement our transformation algorithm, we extended the internal object model that is based on Grafcet by an additional object model that serves as a representation for an IEC 61131-3 function-block. A functionBlock objects contains an interface with all input-, output- and internal variables and a list of assigned methods. Figure 5.5 on the facing page shows the extensions we made to the internal object model in a simplified UML-diagram. The XMLReader class did not need any modifications. The Converter and PLCopenXMLWriter class were replaced by our own implementation.

5.2. Implementation of Converter Class

We created a class named Preprocessor for the transformation of the Grafcet elements into their corresponding Structured Text code fragments according to the transformation rules in section 4.1.

While the declaration of variables in PLCopenXML is done with separate XML-elements, a method’s body consists of only one string. To create and fill the methods of each function block with the code fragments of Grafcet elements, we make extensive
5.2. Implementation of Converter Class

Figure 5.5.: Simplified UML-Diagram of Extensions to the Internal Object Model

use of string concatenation. This is particularly important because we need to make sure that we stick to the evaluation order defined in section 4.2. For our program, that means the order in which we add code fragments to the body member of a method object determines the evaluation order on the PLC.

The object model uses Lists for object collections, e.g., a list of transition objects inside a page object. We use the foreach loop to iterate through these lists and read and process Grafcet elements. To ensure that we stick to the evaluation order, we use LINQ (ref. to section 5.3) to sort the list elements according to their position on the chart. The Grafcet syntax itself only contains information about the connection of elements, but not about their position since such data is not important for the execution of Grafcet. To implement something like a "from top to bottom from left to right" evaluation order, we need to be tool-specific and use the anchor points of the Visio shapes on the chart. This information is stored in the internal object model and can be used to sort the list. For the example of the list of transitions in a page, listing 5.1 is the corresponding instruction that sorts the list from top to bottom and from left to right.

```csharp
functionBlock.Page.TransitionList = sortedTransitionList;
```

Listing 5.1: Example of the sorting of transition objects according to their position

An important requirement for the correct functioning of this sorting strategy is the alignment of elements in the editor:

- Concurrent transitions that succeed the same step need to be horizontally aligned.
- Multiple actions for the same step need to be vertically aligned.
The UML-activity diagram in figure 5.6 shows the transformation process of the control program. It is based on the formal Grafcet definitions in section 2.1.3 on page 12 and the transformation rules depicted in section A.1 on page 63. The subcharts for the transformation of steps and actions are added in the appendix.

Figure 5.6.: Activity Diagram of Transformation Program - Subcharts in Appendix

A great deal of complexity in the program arises from the difference of variable declarations. The Grafcet norm does not require any separate variable declaration. A new variable can just be introduced and used. The type of the variable has to be derived from its usage (e.g., a signal in a continuous action is a bool). Furthermore, the Grafcet
standard does not provide any definition about the available variable types and their semantics. When referring to the type of a variable, the standard uses the expressions `boolean` and `numeric` with no further explanation. We thus assume that the designer of a Grafcet uses the variable types and their semantics of the target implementation, i.e., the IEC 61131-3 for our work. The MS Visio editor was extended by the possibility to declare IEC 61131-3 compliant variables in a separate file and use them in the Grafcet. The IEC 61131-3 requires the declaration of variables for each function block separately. Global variables can be declared in a separate entity, but need to be declared as `VAR_EXTERNAL` if they are used in a function block. The variable file applies to all partial-Grafcets, so the program still needs to identify the variables used in the specific partial-Grafcet. The current implementation extracts all variables from transition conditions and actions during the transformation process, extracts the variable type from the variable file, and adds the variables to the variable list of the function block (second to last step in figure [5.6]). Future work on the editor or even on the Grafcet standard itself should consider including variable declarations for each partial-Grafcet. When constructing a control program, the designer should be aware of the available signals anyway. A variable declaration part (e.g., an additional element) could even simplify the design of a Grafcet because all available signals and variables are visible on the chart.

5.3. Implementation of PLCopenXMLWriter Class

To construct the PLCopenXML file, the function block objects of the internal object model need to be transferred into the corresponding XML-structure. With the Language Integrated Query (LINQ)[27], the Microsoft .NET framework provides a convenient data query language that can be used to read and manipulate XML files. The usage of LINQ in the context of automation is discussed in great detail in [4]. Since the structure of the function block and method objects is already very similar to the PLCopenXML structure, the transformation is straightforward. Each XML element is described by a new instance of the `XStreamingElement` class. Listing 5.2 shows an example of how a method without parameters and return values is transformed into a corresponding PLCopenXML structure.

```csharp
new XElement(ppx + "addData",
    from method in functionBlock.MethodList
    where !method.InputVars.Any() && method.ReturnType == null
    select new XElement(ppx + "data",
        new XAttribute("name", codesys.ToString() + "method"),
        new XAttribute("handleUnknown", "implementation"),
        new XElement(ppx + "Method",
            new XAttribute("name", method.Name),
            new XElement(ppx + "interface"),
            new XElement(ppx + "body",
                new XElement(xhtml + "xhtml", method.Body))
        )
    )
```
5. Prototypical Implementation of Transformation Concept

Listing 5.2: Example of the Construction of a PLCopenXML Element for a Method

5.4. Test and Outlook

The work on this thesis includes the development of the transformation concept of Grafcet charts into Structured Text function blocks and the development of a program that does this transformation automatically. The test of the program follows two main goals:

- Simulate the produced IEC 61131-3 Structured Text code and check the dynamic behavior against the Grafcet specification (check if the concept is right).

- Check if the transformation program fulfills the transformation rules (check if the transformation is done right).

Since all Grafcet elements are read and transformed separately, checking the correct application of the transformation rules means finding a test suite that contains a combination of Grafcet elements such that each transformation rule is applied at least once. The transformation rules are independent from each other, so if this test succeeds, the second test goal can be considered achieved. Referring to the UML activity diagram in figure 5.6, the aim is to find a test suite so that the transformation program traverses each branch of the diagram at least once. The transformation rules for some elements depend on the type of partial-Grafcet to which they belong. Thus, a comprehensive test suite needs to contain a combination of all these elements in all types of partial-Grafcets. We used a set of 9 test cases that include a two-level hierarchical structuring of partial-Grafcets. These test cases consist of each possible combination of the hierarchical elements macrostep, forcing order and enclosure. The graph in figure 5.7 shows the three test cases with a forcing order at the highest level. All tests were successful, i.e., all transformation rules were applied correctly.
The only manual task that needs to be done in the transformation process is the creation of a configuration inside Codesys. Codesys offers an integrated Soft-PLC, named Codesys Win Control V3, that emulates an industrial controller under Windows. A configuration that makes use of this Soft-PLC is the default setting in Codesys. Using this configuration, it was possible to read the Visio-XML files, transform them into PLCopenXML files and import these into the Codesys software development. The resulting control programs can be executed without any further manual modifications.

To validate the dynamic behavior of the transformed code, we used these simulation capabilities and the same set of test cases and focused on the following aspects:

- Transient state evolution during one execution cycle
- Evaluation of stored actions and skipping of continuous actions during one execution cycle
- Correct activation / deactivation of partial-Grafcets with their corresponding hierarchical elements
- Correct state of the program on start of the application

The integrated debugger enables the user to watch the values of all variables, force variables to specified values and execute the program for only one PLC cycle. We used these tools to bring the program to a specific state (i.e., to a specific Grafcet situation), select an input image and execute one cycle. We compared the resulting state of the program and the output image to the Grafcet situation we calculated for the given inputs. All tests we made for the current implementation scope of the program were successful, i.e., the state of the program was equal to the Grafcet situation we calculated.

Appendix A.3 contains yet another example for a Grafcet with hierarchical elements (i.e., a macrostep and an enclosure). It also contains a description of the calculated dynamic behavior and shows the program structure of the transformed IEC 61131-3 program in Codesys.

5.4.1. Events and Timed Grafcets

The current version of the program and the evaluation algorithm do not include events nor time constraints. When introducing these timed properties, the different nature of Grafcet and PLC program evaluation has to be considered again. While in Grafcet, the event \( \uparrow a \) occurs when the signal \( a \) actually changes its value from 0 to 1, the PLC interprets the same event when \( a \) has changed its value at some point between two scan cycles. This can lead to a time difference of a maximum of two cycle duration. The same difference applies for timed actions and constraints. If a timed signal expires just after its value has been evaluated, it will stay active for the duration of one program cycle until its evaluation is done again.

With these differences in mind, [42] shows that it is generally possible to extend the program by event handling and timed variables. The IEC 61131-3 offers standard function blocks for reoccurring tasks like edge detection and time delays. For the detection
of rising and falling edges of events, the R_TRIG and F_TRIG function blocks can be used. They take the signal in question as input and store its value in an internal flag. At the beginning of each cycle, the new value of the signal is compared to its old one and the result of the comparison is written to the function block’s output Q. John and Tiegelkamp mention explicitly that this means that only edges that occur at intervals of at least one program cycle can be detected [22]. Figure 5.8 shows how a function block for rising edge detection is implemented according to the transformation rules. The function block’s input has to be set to the variable / signal a, which could be done anywhere and only once. The output signal of R_TRIG_INST then determines the transition condition. For timed variables, the TON and TOF function blocks offer on- and off-delays for variables. These are used in the same way as the edge detection function blocks.

The implementation of events and timed variables is planned and was left out of the scope of this thesis mostly because of the complexity of extracting variables and their types from signals and actions.

Figure 5.8.: Example of the Usage of Edge Detection Function Blocks
6. Conclusion

While it is common practice in desktop application development, the use of model-driven engineering is still at a very early stage in the domain of automation solution development. Current research is still at the stage of finding appropriate modeling languages and use-case scenarios.

Recent works suggest to use Grafcet, a graphical specification language for the control part of automation systems, to be used as a specification model. The main power of Grafcet is, aside its graphical and thus intuitive way of designing automation control programs, the possibility of structuring a complex control task into smaller units (i.e., partial-Grafcets) that synchronize each other with the use of hierarchical elements. To be able to execute a control program that has been specified in Grafcet on a PLC, it needs to be interpreted as a control program in one of the programming languages defined in the IEC 61131-3 industrial norm. The IEC 61131-3 contains Sequential Function Charts, a programming language that is said to be based on Grafcet. Most previous efforts to find a systematic transformation approach of Grafcet specifications into IEC 61131-3 implementations focus on a transformation into SFC. However, the SFC syntax does not provide equivalent elements for the hierarchical elements of Grafcet. The resulting transformations either ignore hierarchy, or they transform all hierarchical elements into one single Grafcet before the transformation.

The aim of this thesis was to find a transformation approach of Grafcet specifications into an IEC 61131-3 implementation that could preserve the hierarchical structure of the Grafcet. In section 3 we analyzed the potentials of the IEC 61131-3 programming languages as a target language for the transformation. Next to the rather obvious choice of SFC, we also found the usage of Structured Text to be a promising approach. We defined four quality metrics that a transformed program shall meet and used these for a detailed analysis of both languages with a focus on the implementation of hierarchical elements. This analysis revealed that the modeling of Grafcet’s hierarchical elements with SFC results in very complex charts. Furthermore, the semantics and dynamic behavior of SFC are difficult to adapt to Grafcet’s dynamic behavior and are furthermore vendor-dependent. We favored an approach in Structured Text, where the dynamic behavior of Grafcet can be encapsulated in corresponding methods.

On that basis, we developed a Grafcet interpretation algorithm for sequential systems in section 4. Grafcet is a parallel language synchronized on external events. On occurrence of an external event, the state of a Grafcet evolves immediately and all necessary actions are executed simultaneously. A PLC in contrast works in a sequential way and is executed in cycles. That means that a PLC program evolves in each new cycle, while all necessary actions are executed one-by-one. Thus, to be able to interpret a Grafcet specification on a PLC, such an interpretation algorithm is necessary. With the algorithm in place, we defined transformation rules that translate each Grafcet element into a corresponding set of Structured Text code fragments. The resulting control program preserves the hierarchical structure of partial-Grafcets with the use of corresponding IEC 61131-3 function blocks.
6. Conclusion

To be able to test the developed transformation concept, we built a prototypical transformation program in section 5. The prototype was built on top of an existing program for the transformation of Grafcet into SFC without hierarchical elements. The program makes use of a Grafcet editor based on Microsoft Visio and outputs the transformed IEC 61131-3 Structured Text program using the PLCopenXML file format. We were able to import this file into the widely used automation software development environment Codesys, where we could test the program on a Soft-PLC. In a first iteration, we tested if the program implements all transition rules correctly. Secondly, we defined a set of test cases to check the dynamic behavior of the transformed IEC 61131-3 control program against the manually computed behavior of the corresponding Grafcet. All tests were successful.

6.1. Future Work

The validation of the current transformation concept has been done with a series of tests that focused on a number of dynamic properties, but were selected rather intuitively. It is desirable to undertake a structured test approach to be able to have a qualitative and quantitative measure about the concept’s correctness. Another interesting approach in terms of concept validation would be the use of model checking techniques on the basis of the interpretation of Grafcet as a control interpreted Petri net as in [26].

Future work on the implementation scope of the prototype should focus on the integration of events and timed properties as in section 5.4.1. Another important aspect is the evaluation order of Grafcet elements in the transformed Structured Text code. Currently, the elements of a partial-Grafcet are evaluated according to their position on the chart, while the order of the partial-Grafcets themselves is determined by their corresponding VisioID which is related to their creation timestamp. Especially the latter is a rather implicit property and not obvious to the designer. It would be desirable to be able to define an explicit ordering within the Grafcet editor, e.g., by introducing a priority measure.

One main advantage of model-driven engineering using formal methods is the possibility to use static code analysis, i.e., applying mathematical methods to the model to check for general safety properties (e.g., deadlocks, unbounded loops) and user-defined constraints and properties. The translation of Grafcet into CIPN in [42] enables the usage of existing CIPN static code analysis tools. These could be used to check if the design requirements that we defined in section 4.1 are met before a transformation is undertaken.

Currently, all tests of the transformed IEC 61131-3 implementation are done in Codesys. Some aspects of the transformation, i.e., the structure of the PLCopenXML file, are tailored to that tool. It can be assumed that new releases of other control software development environments will include the object-oriented extensions of function blocks of the new IEC 61131-3 and also implement a PLCopenXML import / export functionality. Changing the structure of the resulting PLCopenXML file of the prototype is easy to achieve, since only one class needs to be altered / extended. Thus, the transformation
6.1. Future Work

The prototype should be extended to support all new releases of other software development environments.

In the long term, the transformation concept could even be the basis of an integrated Grafcet editor within the PLC development environments. Since Grafcet is part of the curriculum of technicians in Germany, it is desirable to adopt Grafcet into an integrated engineering process.

An interesting extension of the transformation program would be the backwards transformation of the IEC 61131-3 program back into a Grafcet representation, as we describe in section 6.1.1.

6.1.1. Possibility of Bidirectional Transformation

One of the biggest advantages of the Structured Text approach over most of the other discussed ways to transform Grafcet into an IEC 61131-3 implementation is the possibility of a bi-directional transformation approach, while preserving the original structure of the Grafcet. The process of transforming an existing implementation back to a specification model is called reverse engineering and is part of ongoing research because of the wide use of different and sometimes even ancient programming languages in the automation domain [3]. The necessity for such a backwards transformation results from the fact that minor changes to the control program are usually done directly in the code by technicians. These changes typically include additional transition conditions or the change of output values, caused by forgotten or failed actors and sensors in the plant. When the original model is used as a basis for the development of other or additional control programs, it is important that such small modifications are updated in the model as well. An automated transformation approach facilitates these updates and prevents the need for manual comparison of current program and original model. Referring to the waterfall model in the introduction (figure 1.1 on page 1), a bidirectional transformation approach enables the system model (i.e., the Grafcet specification) to conduct the SLDC along all stages, including deployment and maintenance.

Transforming a state machine back into a Grafcet is a straightforward task. The easiest way would be to interpret each state as a step, each transition function as a transition condition and each output as a continuous action. However, constructing the original Grafcet with the hierarchical structure from a state machine is not possible. The normalization approach in [42] makes it very difficult to transform the SFC implementation back to Grafcet. The transformed implementation does not include information about the number of partial-Grafcets that have been merged into one global Grafcet. This means that one does not know how many elements need to be extracted from the global Grafcet in order to gain the previous hierarchical structure of the underlying Grafcet before the transformation. While the beginning and end of Macrostep expansions are still identifiable via the unique names of the entry- and exit-step, remodeling enclosed and forced Grafcets out of the global Grafcet is a very difficult task.

For the Structured Text approach, only the interfaces and some methods need to be evaluated to construct the original Grafcet. The following list is an informal description that could be the basis of a transformation algorithm. It anticipates the naming of ele-
ments from the prototypical transformation program in section 5.2.

For each function block...

1. create a partial-Grafcet with the name of the function block.

2. read the interface. Create a step for each variable that starts with a capital S, M or E. Create a transition variable for each variable that starts with a capital T.

3. read the method "evaluateTransitions" line by line. Find the transition with name "left-hand-side of the expression", add the right-hand-side as its transition condition.

4. read the method "stateEvolution" if-statement per if-statement. Find the transition with name "if-condition". Add all steps that are set to 0 as preceding and all steps that are set to 1 as succeeding to the transition.

5. read the method "continuousActions" if-statement per if-statement. Create a continuous action. Link the action to the step with name "if-condition". Compare the body to transformation rules 12 to 15. If one of the rules applies, change the action into the respective forcing order. Else, add the body of the if-statement to the action’s body.

6. read the method "storedActions" if-statement per if-statement. If the body looks like transition rule 4 or 5 and the if-condition contains a capital M, find the corresponding step with name "if-condition" and change it into a macrostep. If the body looks like transition rule 4 or 5 and the body contains a capital S, find the corresponding step with name "if-condition" and change it into an enclosure. In all other cases, create a stored action and link it to the step with name "if-condition" and add the body of the if-statement to the action’s body.

7. If the function block’s name contains a capital X (i.e., it is an enclosed Grafcet), read the method "initializeAndStop" line per line. Mark all steps that are set to 1 with an asterisk in the Grafcet.

The order of the lines and if-statements in the respective method is determined by the position of these elements in the Grafcet. If the information about this order is extracted during the backwards transformation, even the relative position of concurrent transitions and multiple actions can be restored.

It has to be noted that direct changes to the code only make sense for minor modifications like additional transformation conditions and changes in actions. Other changes like the introduction of new steps require modifications in different parts in the code, which is very error-prone. For example, the introduction of a new initial step in a forced Grafcet requires the introduction of a new method and a new variable, and the modification of two methods. In this case, it would make sense to use the automated transformation of the implementation back to a Grafcet, make the change in the Grafcet and transform it back to an implementation.
Appendices
A. Appendix

A.1. Transformation Rules

As described in section 3.2, the transformation is based on one-to-one correspondence of Grafcet-elements to Structured Text code fragments. The following table represents these transformation rules. After initializing and creating the code shells (see appendix A.1.2), each Grafcet-element is analyzed and the corresponding code fragments are added to the specific position in the shell which is tagged in green. The transformation rules of certain Grafcet elements depend on the type of their partial-Grafcet. A basic set of transformation rules applies to all types of partial-Grafcets and is given in appendix A.1. The transformation rules given in the following tables apply in addition to the set of basic rules. If a transformation rule for an element differs from the set of basic rules, it is marked with \textit{overwrite}. Note that the IEC 61131-3 suggests to write Structured Text constructs in capital letters, which has not been done here for the sake of readability.

A.1.1. Transformation of Grafcet Elements

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafcet Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Step</td>
<td>\texttt{S1: bool; // VAR}</td>
</tr>
</tbody>
</table>
| 2   | Initial Step   | \begin{align*} S1: \text{bool} & := \text{true}; // VAR \\
|     |                | \texttt{------------------} \\
|     |                | \texttt{public method activateState1} \\
|     |                | \texttt{S1 := 1;} \\
|     |                | \texttt{end_method}           \end{align*} |
| 3   | Macrostep      | \begin{align*} M2: \text{bool}; // VAR \\
|     |                | \texttt{------------------} \\
|     |                | \texttt{if M2 then // storedActions} \\
|     |                | \texttt{fb\_M2\_activate();} \\
|     |                | \texttt{else} \\
|     |                | \texttt{fb\_M2\_initializeAndStop();} \\
|     |                | \texttt{end\_if;}            \end{align*} |
| 4   | Enclosure      | \begin{align*} S8: \text{bool}; // VAR \\
|     |                | \texttt{------------------} \\
|     |                | \texttt{if S8 then // storedActions} \\
|     |                | \texttt{fb\_X8\_activate();} \\
|     |                | \texttt{else} \\
|     |                | \texttt{fb\_X8\_initializeAndStop();} \\
<p>|     |                | \texttt{end_if;}            \end{align*} |</p>
<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafset Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Initial Enclosure</td>
<td>S9 := true;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if S9 then // storedActions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fb_X9.activate();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fb_X9.initializeAndStop();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end_if;</td>
</tr>
<tr>
<td>6</td>
<td>Transition</td>
<td>T22: bool; // VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT22: bool;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T22 := cond; // evaluateTransitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT22 := T22 and S22;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// evaluateFireableTransitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if FT22 then // stateEvolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stableState := false;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S22 := 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S23 := 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end_if;</td>
</tr>
<tr>
<td>7</td>
<td>Synchronization</td>
<td>T10: bool; // VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT10: bool;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T10 := 1; // evaluateTransitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT10 := T10 and S1 and S2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// evaluateFireableTransitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if FT10 then // stateEvolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stableState := false;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1 := 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S2 := 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S3 := 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S4 := 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S5 := 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end_if;</td>
</tr>
<tr>
<td>8</td>
<td>Transition after Macrostep</td>
<td>T22: bool; // VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T22 := cond; // evaluateTransitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT22 := T22 and S22 and fb_M2.GetFinishedFlag();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// evaluateFireableTransitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if FT22 then // stateEvolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stableState := false;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M2 := 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S24 := 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end_if;</td>
</tr>
</tbody>
</table>
### A.1. Transformation Rules

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafcet Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
</table>
| 9   | Continuous Action | \[
signal_1 := 0; \ // \text{pullDownSignals}
-------------
\text{if } S1 \ \text{then} \ // \text{continuousActions}
\quad signal_1 := 1;
\text{end_if;}
\] |
| 10  | Continuous Action with Condition | \[
signal_1 := 0; \ // \text{pullDownSignals}
-------------
\text{if } S1 \ \text{then} \ // \text{continuousActions}
\quad signal_1 := 1 \ \text{and } d;
\text{end_if;}
\] |
| 11  | Stored Action | \[
\text{if } S1 \ \text{then} \ // \text{storedActions}
\quad k := k + 1;
\text{end_if;}
\] |
| 12  | Forcing Specific Situation | \[
\text{if } S1 \ \text{then} \ // \text{continuousActions}
\quad \text{fb}_G2.\text{deactivateAllSteps();}
\quad \text{fb}_G2.\text{setForcedFlag(TRUE);}
\quad \text{fb}_G2.\text{activateStateS21();}
\quad \text{fb}_G2.\text{activateStateS23();}
\text{end_if;}
\] |
| 13  | Forcing Empty Situation | \[
\text{if } S1 \ \text{then} \ // \text{continuousActions}
\quad \text{fb}_G2.\text{deactivateAllSteps();}
\quad \text{fb}_G2.\text{setForcedFlag(TRUE);}
\text{end_if;}
\] |
| 14  | Forcing Current Situation | \[
\text{if } S1 \ \text{then} \ // \text{continuousActions}
\quad \text{fb}_G2.\text{setForcedFlag(TRUE);}
\text{end_if;}
\] |
| 15  | Forcing Initial Situation | \[
\text{if } S1 \ \text{then} \ // \text{continuousActions}
\quad \text{fb}_G2.\text{deactivateAllSteps();}
\quad \text{fb}_G2.\text{setForcedFlag(TRUE);}
\quad \text{fb}_G2.\text{activateInitSituation();}
\text{end_if;}
\] |

#### Table A.1.: Basic Transformation Rules

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafcet Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
</table>
| 16  | Step | \[
public method activateStateS1
\quad S1 := 1;
\text{end_method}
-------------
\text{if } S1 \ \text{then} \ // \text{deactivateAllSteps}
\] |
### A. Appendix

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafcet Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Initial Step</td>
<td>public method activateStateS1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1 := 1; end_method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1 := 0; // deactivateAllSteps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1 := 1; // activateInitSituation</td>
</tr>
<tr>
<td>18</td>
<td>Macrostep</td>
<td>public method activateStateM2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M2 := 1; end_method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M2 := 0; // deactivateAllSteps</td>
</tr>
<tr>
<td>19</td>
<td>Enclosure</td>
<td>public method activateStateS8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S8 := 1; end_method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S8 := 0; // deactivateAllSteps</td>
</tr>
<tr>
<td>20</td>
<td>Initial Enclosure</td>
<td>public method activateStateS9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S9 := 1; end_method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S9 := 0; // deactivateAllSteps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S9 := 1; // activateInitSituation</td>
</tr>
</tbody>
</table>

Table A.2.: Additional Transformation Rules in a Classical Chart

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafcet Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Step</td>
<td>S1 := 0; // initializeAndStop</td>
</tr>
<tr>
<td>22</td>
<td>Initial Step</td>
<td>S1 := 0; // initializeAndStop</td>
</tr>
<tr>
<td>23</td>
<td>Macrostep</td>
<td>M2 := 0; // initializeAndStop</td>
</tr>
<tr>
<td>24</td>
<td>Enclosure</td>
<td>S8 := 0; // initializeAndStop</td>
</tr>
<tr>
<td>25</td>
<td>Initial Enclosure</td>
<td>S9 := 0; // initializeAndStop</td>
</tr>
<tr>
<td>26</td>
<td>Enclosed Step</td>
<td>S11 : bool; // VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S11 := 1; // initializeAndStop</td>
</tr>
</tbody>
</table>

66
A.1. Transformation Rules

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafcet Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Enclosed Initial Step</td>
<td>S11 : bool := true; // VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-----------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S11 := 1; // initializeAndStop</td>
</tr>
</tbody>
</table>

Table A.3.: Additional Transformation Rules in an Enclosed Grafcet

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Grafcet Element</th>
<th>Representation in IEC 61131-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Entry Step</td>
<td>E1 : bool := true; // VAR overwrite</td>
</tr>
</tbody>
</table>

Table A.4.: Additional Transformation Rules in a Macrostep Expansion

A.1.2. Code Shell for Partial Grafcets

Each partial-Grafcet is transformed into one function block. Depending on the nature of the partial-Grafcet (i.e., classical, enclosed, macrostep expansion), the corresponding function block has to offer a number of methods which are filled during the transformation. In this work, the structure of a function block with empty methods is called shell.
Listing A.29: Shell for a Macrostep Expansion

```
function_block functionBlockNAME //replace NAME by name of partial Grafcet

var_external
// input/output signals
// globally declared
end_var_external
var
// put steps here
// put transitions here
// internal triggers
finishedFlag: bool;
active: bool;
end_var

// communication methods
public method initializeAndStop: void
if active then
  pullDownSignals( );
  E* := 0 ;
  E* := 1;
  active:= false;
  finishedFlag := false;
end_if;
end_method

public method activate: void
active:= true;
end_method

public method getFinishedFlag: bool
getFinishedFlag := finishedFlag;
end_method

// transition conditions
public method evaluateTransitions: void
// put content here
end_method

// firing conditions
public method evaluateFireableTransitions: void
// put content here
end_method

// transient state evolution
public method stateEvolution: bool
var_input
  stableState: bool;
end_var
if active then
  // put content here
end_if;
stateEvolution := stableState;
end_method

// continuous action evaluation
public method pullDownSignals: void
// put content here
end_method

public method continuousActions: void
if active then
  // put content here
end_if;
end_method

// stored action evaluation
public method storedActions: void
if active then
  if S* then
    finishedFlag := 1;
  end_if;
  // put content here
end_if;
end_method

end_function_block
```
A.1. Transformation Rules

function_block functionBlockForcing
  var_external
  // input/output signals
  // globally declared
  end_var
  var
  // put steps here
  // put transitions here
  // internal triggers
  forcedFlag: bool := false;
  end_var
  // communication methods
  public method deactivateAllSteps: void
  // deactivate all steps here
  end_method
  public method activateInitSituation: void
  // set all initial steps to true here
  end_method
  public method setForcedFlag: void
    var_input
    flag: bool;
    end_var
    forcedFlag := flag;
  end_method
  public method getForcedFlag: bool
    getForcedFlag := forcedFlag;
  end_method
  // transition conditions
  public method evaluateTransitions: void
  // put content here
  end_method
  // firing conditions
  public method evaluateFireableTransitions: void
  // put content here
  end_method
  // transient state evolution
  public method stateEvolution: bool
    var_input
    stableState: bool;
    end_var
    if not forcedFlag then
      // put content here
    end_if;
    stateEvolution := stableState;
  end_method
  // continuous action evaluation
  public method pullDownSignals: void
  // put content here
  end_method
  public method continuousActions: void
  // put content here
  end_method
  // stored action evaluation
  public method storedActions: void
  // put content here
  end_method
end_function_block

Listing A.30: Shell for a Classical Chart
function_block functionBlockEnclosure
var_external
// input/output signals
// globally declared
end_var
var
// put steps here
// put transitions here
// internal triggers
active: bool := false;
end_var
// communication methods
public method initializeAndStop: void
if active then
// deactivate all steps here
pullDownSignals();
// activate all enclosed steps here
active := false;
end_if;
end_method
public method activate: void
active := true;
end_method
// transition conditions
public method evaluateTransitions: void
// put content here
end_method
// firing conditions
public method evaluateFireableTransitions: void
// put content here
end_method
// transient state evolution
public method stateEvolution: bool
var_input
stableState: bool;
end_var
if active then
// put content here
end_if;
stateEvolution := stableState;
end_method
// continuous action evaluation
public method pullDownSignals: void
// put content here
end_method
public method continuousActions: void
if active then
// put content here
end_if;
end_method
// stored action evaluation
public method storedActions: void
if active then
// put content here
end_if;
end_method
end_function_block
A.2. Activity Diagrams of Transformation Program

Figure A.1.: Activity Diagram of Transformation Program - Subchart Process Steps
Figure A.2.: Activity Diagram of Transformation Program - Subchart Process Actions
A.3. Extended Example of the Transformation

Example A.3.1. Consider the Grafcet in figure A.3. The main Grafcet GMain includes a macrostep and an enclosure, with M1 and X3/G2 as the corresponding partial-Grafcets. Figure A.4 on page 75 shows a screenshot of the IEC 61131-3 control program in Codesys after the transformation of the Grafcet and the import of the PLCopenXML file. Table A.5 on the following page shows the test cases we used to compare the dynamic behavior of the IEC 61131-3 program to the results we computed for the Grafcet.

The initial situation is given by the initial step 1 in GMain and arbitrary chosen default values for the numeric input variables INT1, INT2 and the numeric output variable AO_2. All other boolean input and output signals are false by default (Cycle 0).
### Table A.5: Grafcet Evolution Example

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Active Inputs</th>
<th>INT1</th>
<th>INT2</th>
<th>Active Steps</th>
<th>Active Outputs</th>
<th>AO_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>DI_1</td>
<td>3</td>
<td>6</td>
<td>2 ,M1, 12</td>
<td>DO_14</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>DI_2, DI_15</td>
<td>3</td>
<td>6</td>
<td>3 ,M1, S1, 7</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>DI_10</td>
<td>3</td>
<td>6</td>
<td>3 ,M1, S1, 8</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>DI_11</td>
<td>3</td>
<td>6</td>
<td>3 ,M1, S1, 6</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DI_3</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

In Cycle 1, transient behavior can be observed. The activation of the input DI_1 leads to the firing of transition t1 and the activation of the steps 2 and M1. The latter activates the macrostep expansion and thus step E1. Because transition t22 is fireable, it fires immediately, deactivates step E1 and activates step 12. Step E1 has thus been part of an unstable situation.

In the following, we only give short comments on the dynamic behavior of the Grafcet. Comprehensive information can be derived from table [A.5].

In Cycle 2, the activation of the enclosure 3 activates the corresponding partial-Grafcet with the enclosed step 7. The stored action associated to step 7 sets AO_2 to 3. The deactivation of step 12 also deactivates the output DO_14 of the corresponding continuous action.

While cycle 3 is a normal evolution step, cycle 4 leads to transient evolution again. When transition t15 fires, transition t11 will fire immediately afterwards because it has been enabled all the time. The resulting activation of step 6 leads to the execution of the corresponding stored action which sets AO_2 to 1.

In cycle 5, transition t3 can fire because step 3 and M1 are enabled and the macrostep expansion of M1 has reached the exit step S1. After the firing, all steps of the enclosed partial-Grafcet X3/G2 and the macrostep expansion M1 are inactive. Nevertheless, the outputs that have been manipulated by stored actions in one of these partial-Grafcets before keep their current value.

The test cases with the input sequence of table [A.5] resulted in the same dynamic behavior of the transformed IEC 61131-3 implementation as it was computed in the table.
A.3. Extended Example of the Transformation

Figure A.4.: Transformed IEC 61131-3 Control Program of the Grafcet in Figure A.3 after Codesys Import
### Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPN</td>
<td>Control Interpreted Petri Net</td>
</tr>
<tr>
<td>Configuration</td>
<td>PLC system</td>
</tr>
<tr>
<td>FBD</td>
<td>Function Block Diagram</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IL</td>
<td>Instruction List</td>
</tr>
<tr>
<td>LD</td>
<td>Ladder Diagram</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controller</td>
</tr>
<tr>
<td>POU</td>
<td>Program Organization Unit</td>
</tr>
<tr>
<td>Resource</td>
<td>Processing unit inside a configuration</td>
</tr>
<tr>
<td>SDLC</td>
<td>Systems Development Lifecycle</td>
</tr>
<tr>
<td>SFC</td>
<td>Sequential Function Charts</td>
</tr>
<tr>
<td>Shell</td>
<td>Function block that includes a set of empty methods</td>
</tr>
<tr>
<td>SIPN</td>
<td>Signal Interpreted Petri Nets</td>
</tr>
<tr>
<td>ST</td>
<td>Structured Text</td>
</tr>
<tr>
<td>Task</td>
<td>Assigns runtime properties to POU</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
Bibliography

[1] 3S-Smart Software Solutions. CODESYS – industrial IEC 61131-3 PLC program-
ing. 2015-03-10. URL: http://www.codesys.com/ (visited on 06/13/2015).

org/o.red.c/organisation.html (visited on 06/13/2015).

[3] Mohammed Bani Younis. Re-engineering approach for PLC programs based on

[4] Mike Barth and Alexander Fay. ‘Efficient use of data exchange formats in engi-
neering projects by means of language integrated queries — Engineers LINQ to
XML’. In: IECON 2010 - 36th Annual Conference of IEEE Industrial Electronics,
pp. 1335–1340. DOI: 10.1109/IECON.2010.5675490

[5] Nanette Bauer et al. ‘A Unifying Semantics for Sequential Function Charts’. In:
Integration of Software Specification Techniques for Applications in Engineering.
Ed. by David Hutchison et al. Vol. 3147. Lecture Notes in Computer Science.
Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 400–418. ISBN: 978-3-
540-23135-6.

in Produktion, Automatisierung und Logistik: Anwendung · Technologien · Migra-

microcontrollers’. In: Assembly Automation 28.1 (2008), pp. 55–60. ISSN: 0144-
5154. DOI: 10.1108/01445150810849019

[8] D. L’Her, P. Le Parc, and L. Marcé. ‘Modeling And Proving Grafcets With Tran-
sition Systems’. In: In 2nd AMAST workshop on Real-Time Systems and Project

catia/capabilities/embedded-systems/controlbuild/ (visited on 06/13/2015).

[10] René David. ‘Grafcet: a powerful tool for specification of logic controllers’. In:
IEEE Transactions on Control Systems Technology 3.3 (1995), pp. 253–268. DOI:
10.1109/87.406973

ebrary.com/lib/alltitles/docDetail.action?docID=10430267

[12] Deutsches Institut für Normung e.V. Speicherprogrammierbare Steuerungen – DIN
EN 61131-3 Beiblatt 1 - Leitlinien für die Anwendung und Implementierung von


