Model-Based Design of Program Organization Units Using Synchronous Languages

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Marcel Christian Werner

Date of Defense:	July 4, 2025
Dean:	Prof. Dr. Christoph Garth
Reviewer:	Prof. Dr. Klaus Schneider
Reviewer:	Prof. Dr. Reinhard von Hanxleden

DE-386

Abstract

Programmable Logic Controllers (PLCs) are typically applied in industrial environments with real-time requirements. In many cases, PLC development is based on standards defined by the International Electrotechnical Commission (IEC), in particular IEC 61131-3. This standard describes various languages for developing Program Organization Units (POUs), including the textual language Structured Text (ST) and graphical Function Block Diagrams (FBDs), two languages that have become widely accepted and often used in real-world applications. POUs are typically used over a long period of time and extended incrementally. As a result, their structural complexity and the associated challenges such as maintainability and readability tend to increase. In addition, safety-critical applications require formal verification, which can be achieved by translating IEC 61131-3 POUs into formal models. This often results in domain-specific models for verification purposes only. Furthermore, the transition to other development approaches, such as model-based design or changes to hardware that is not IEC 61131-3 compliant, often requires that existing IEC 61131-3 POUs be partially or completely designed from scratch.

In contrast, synchronous languages have proven to be efficient for the modeling and formal verification of reactive real-time systems in both research and industrial applications. Therefore, this thesis introduces several detailed transformations to translate existing ST- and FBD-based POUs into semantically equivalent synchronous models. These transformations allow POUs to be reused in a model-based design approach that supports formal verification. Furthermore, a formal methods-based optimization approach is introduced, which significantly reduces (on average) the structural complexity of real-world data-flow models such as FBDs. It is also shown how hierarchical statecharts can be derived from ST- and FBD-based models to provide an alternative graphical representation of the control flow. The correctness of these approaches is analyzed both theoretically, based on the syntax and formal semantics of the languages, and practically, based on appropriate IEC 61131-3 POUs, by integrating the transformation and optimization approaches into *PLCreX*, an application developed as part of this research.

Acknowledgement

I would like to thank all the people who have supported me in the challenge of balancing research activities with personal commitments to family, friends, teaching, and a full-time job. In particular, I would like to thank Prof. Dr. Klaus Schneider, who supported me especially in hectic phases such as meeting submission deadlines. I would also like to thank him for his valuable guidance and constructive feedback on my research over the past four years. Furthermore, I would like to thank Prof. Dr. Reinhard von Hanxleden for serving as my second assessor. I also want to thank Prof. Dr. Sebastian Michel for chairing the doctoral committee and Prof. Dr. Christoph Grimm for being a member of the doctoral committee. Additionally, I would like to acknowledge Dr. Manuel Gesell for his valuable feedback during the final stages of my thesis. His willingness to review my thesis in his free time is exceptional, and I am very grateful for his efforts. I would also like to thank Marita Stuppy and Sabine Owens for their support in administrative topics. Finally, I would like to express my deepest gratitude to my friends, and especially to my partner and son, for their acceptance and support of my personal objectives, especially during the most challenging phases of this thesis. Their contributions, whether large or small, have been invaluable and I am very grateful for their support.

July 4, 2025, Marcel Christian Werner

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

International standards, such as those set by the International Electrotechnical Commission (IEC), in particular IEC 61131-3 [GDV14], describe the development of software applications for industrial control systems with real-time requirements, such as Programmable Logic Controllers (PLCs). IEC 61131-3 describes a set of languages for the development of PLC software applications, which are organized in so-called Program Organization Units (POUs). In addition to common bitwise and arithmetic operators, in many cases POUs contain predefined functions and function blocks from external libraries whose internal behavior is unknown. In real-world applications, graphical Function Block Diagrams (FBDs) and textual Structured Text (ST) models are widely accepted and used languages for developing POUs. FBDs are characterized by a graphical data flow notation, typically executed from left to right. They often allow manual positioning and graphical linking of components such as blocks and variables on a POU worksheet using visual edges. In contrast, ST models allow the development of POUs in a textual, imperative language derived from *Pascal* [Wir71]. Over the past two decades, in addition to these traditional approaches, further methods have been explored with the goal of streamlining the development of POUs through the use of modeling techniques such as the Unified Modeling Language[1] (UML) [WV04]. This extended modeling approach is reflected in modern PLC development environments, such as provided by *CODESYS* [WV09]. Another trend in modern engineering tools is code generation for POUs as part of a model-based design approach, such as provided by *Simulink* [BAV08].

The use of UML and integration with model-based design approaches provide advanced methods for simplifying the development process for new POUs. However, rather than developing POUs from scratch, it is more common in real-world applications to extend existing POUs with additional functionality or logic during their lifecycle. These incremental extensions typically tend to increase the structural complexity of the POUs. While text-based ST models result in more lines of source code, graphical FBDs result in a greater number

[1] `https://www.uml.org/`

of graphical components. The readability of FBDs can be affected by this increased complexity, as the number of components and connecting edges can be perceived as chaotic, making it more difficult to understand. Another challenge is that formal verification techniques for POUs are essential, especially in safety-critical applications. This requires the translation of existing POUs into formal models, which is the bulk of research in the area of IEC 61131-3 applications. Since the goal is verification rather than functional reuse, in most approaches the translations of existing POUs into formal models result in domain-specific models. Overall, changes to industrial POUs can be very time consuming and costly. These changes also require that the IEC 61131-3 engineering approach be maintained. In real-world applications, this means that a change in hardware that is not IEC 61131-3 compliant, or a change in engineering approach, often results in scenarios where existing engineering efforts cannot be reused (or only partially reused) and software applications have to be developed from scratch.

In contrast, synchronous languages [BB91; Ben+03] have proven effective for the design and formal verification of reactive real-time systems in research and in isolated commercial model-based development environments, such as *SCADE* [Le +11]. Two publicly available frameworks, which have established a reputation in academia for model-based design and formal verification are *Averest*[2] [SS06] and the *KIELER*[3] [Kas+24] project. *Averest* uses as an input model the imperative synchronous language *Quartz* [SB16], which is derived from *Esterel* [BG92]. *KIELER* uses several input models, including the Sequentially Constructive Language (SCL) and the Sequentially Constructive Statecharts (SCCharts) [Han+14], where sequentially constructive concurrency is a conservative extension of the classical Synchronous Model of Computation (SMoC) [Han+13].

1.1. Contributions

Based on the challenges in real-world applications and the established reputation of synchronous languages, the following hypotheses **H1**, **H2**, and **H3** are formulated:

H1: ST-based and FBD-based POUs can be translated into synchronous models without losing the original runtime behavior or level of abstraction (in terms of variables and structure). This translation enables reuse in model-based design, supports formal verification, and allows intuitive post-translation modification.

H2: The structural complexity of data-flow models in real-world applications is often greater than the logic requires (in terms of variable accesses, operators, and edges). This complexity can be reduced while preserving the original functionality.

[2] `http://www.averest.org/`
[3] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview`

H3: ST-based and FBD-based *Quartz* models can be synthesized into graphical, control-flow oriented SCCharts. Specifically, applying a pattern-based transformation process to FBD-based *Quartz* models yields hierarchical, control-flow oriented SCCharts (if matching patterns are available), providing an alternative control-flow view for system analysis.

To answer these hypotheses, this thesis contributes to the field of reuse of existing IEC 61131-3 POUs in model-based design using synchronous languages and their optimization within the synchronous paradigm. The contributions can be divided into the following three categories derived from the hypotheses:

1. **H1: Model-Based Design of Program Organization Units**

   The first contribution of this thesis focuses on a possible reuse of textual ST models Ω_{st}^{φ} in model-based design using synchronous models by introducing a detailed model transformation from $\omega_{st}^{\varphi} \in \Omega_{st}^{\varphi}$ to a corresponding *Quartz* model $\omega_{qrz} \in \Omega_{qrz}$ and SCL model $\omega_{scl} \in \Omega_{scl}$, as shown in Figure 1.1. The correctness of the transformations is proved by theoretical reasoning, and the theoretical results are evaluated with real-world and self-defined ST models.

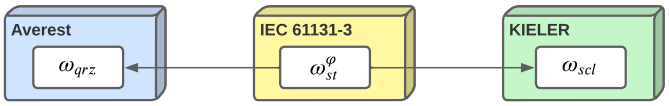


Figure 1.1.: *Contribution: ST-to-*Quartz*/SCL transformation*

   The second contribution focuses on a possible reuse of graphical FBDs Ω_{fbd}^{φ} in model-based design using synchronous models by introducing a detailed model transformation from $\omega_{fbd}^{\varphi} \in \Omega_{fbd}^{\varphi}$ to a corresponding data-flow oriented *Quartz* model $\omega_{qrz} \in \Omega_{qrz}$ and data-flow oriented *SCChart* $\omega_{scd} \in \Omega_{scd}$, as shown in Figure 1.2. The correctness of the transformations is proved by theoretical reasoning, and the theoretical results are evaluated with real-world and self-defined FBDs.

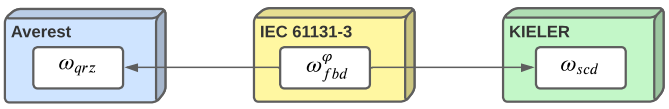


Figure 1.2.: *Contribution: FBD-to-*Quartz*/SCChart transformation*

2. **H2: Formal Methods-Based Optimization of Data-Flow Models**

   The third contribution focuses on a formal methods-based optimization of data-flow models by introducing a configurable optimization process of $\omega_d \in \Omega_d$, as shown in Figure 1.3. The correctness of the optimization process is proved by theoretical reasoning and ensured by integrated equivalence checking. The optimization potential in real-world applications is evaluated by experimental results.

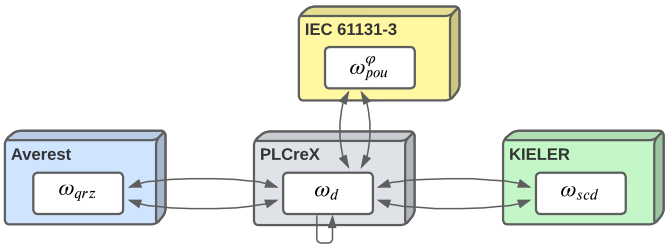


Figure 1.3.: *Contribution: Data-Flow Optimization*

3. **H3: Control-Flow Oriented SCCharts of *Quartz* Models**

   The fourth contribution focuses on the synthesis of *Quartz* models Ω_{qrz} into control-flow oriented SCCharts Ω_{scc} by introducing a pattern-based *Quartz* code refactoring intended for data-flow oriented *Quartz* models, and a subsequent model transformation from $\omega_{qrz} \in \Omega_{qrz}$ to $\omega_{scc} \in \Omega_{scc}$, as shown in Figure 1.4. The correctness of the transformation is proved by theoretical reasoning, and the theoretical results are evaluated with real-world and self-defined *Quartz* models.

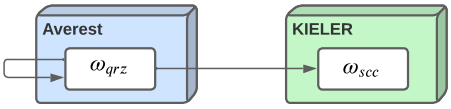


Figure 1.4.: *Contribution:* Quartz-*to-SCChart Transformation*

These approaches have motivated the development of *PLCreX*[4], a project for simplification, transformation, analysis and validation of PLCs, which represents a further contribution of this thesis.

1.2. Related Work

There are various approaches to integrate model-based design concepts into the IEC 61131 development process. For example, Thramboulidis and Frey [TF11] explored the use of IEC 61499, UML[5], and SysML[6] to increase the effectiveness of the IEC 61131 development processes. This extended modeling approach is reflected in modern PLC development environments, such as provided by *CODESYS* [WV09]. Furthermore, automatic translation of UML models into IEC 61131-3 models has been proposed in [WV04] and the translation of *Matlab/Simulink/Stateflow* models into IEC 61131-3 models in [BAV08].

In the context of model-based testing, an application of model-based design [ZSM11], Rösch et al. [Rös+15] stated that most of the transformations of ST-based and FBD-based POUs to formal models are performed for verification

[4] `https://github.com/marwern/PLCreX`
[5] `https://www.uml.org/`
[6] `https://sysml.org/`

purposes. This focus of research continues to this day. Various methods have been developed to enable the application of formal verification techniques such as *model checking* [CGP01] and *theorem proving* [Har08]. For example, the translation of FBDs into the *Prototype Verification System* [ORS92] has been addressed in [New+16] and [New+18]. Transformation rules for converting IEC 61131-3 FBDs to *UPPAAL* models [BDL04] were introduced in [STF12]. The application of *UPPAL* for the verification of Continuous Function Charts (CFCs), which can be interpreted as extended FBDs [Bia16], was introduced in [WFV09]. A mutation-based test generation using *UPPAAL* was explored in [Eno+16]. Furthermore, Pavlovic et al. [PE10] outlined a process for verifying FBDs using the *NuSMV* model checker [Cim+00]. The translation of POUs to an intermediate model compatible with the *nuXmv* model checker [Cav+14] was introduced in [Fer+15]. A regression verification technique by translating POUs into the SMV language was proposed in [Bec+15]. Barbosa and Déharbe [BD12] have focused on the *B Method* [CM03] and presented an approach to automatically translate IEC 61131-3 POUs into *B machines*. Additionally, a theorem-prover supported verification was suggested in [VK02]. The formalization of POUs has also been addressed using proof assistants. For example, Blech and Biha [BB13] have formalized the semantics of the IEC 61131-3 languages in the formal proof management system *Coq* [Ber16]. Yanhong et al. [Hua+19] have defined the operational semantics of ST models within the *K framework*[7]. Apart from this, Darvas et al. [DMB16] stated that the ST language can efficiently and conveniently represent all PLC languages for formal verification purposes. In summary, these approaches highlight the importance of translating IEC 61131-3 POUs into formal models to apply advanced verification techniques.

There are further approaches that have been developed to enhance the testing and verification of IEC 61131-3 POUs. For example, Jee et al. [Jee+09] have introduced a data-flow-based structural testing technique for FBDs. Xiong et al. [Xio+20] have presented a specification-mining-based verification method tailored for IEC 61131-3 POUs. Apart from this, several frameworks and tools have been developed to enhance the automation, verification, and analysis processes. For example, *PLCAutoTester* [Shi+24] represents an unit test case generation framework for ST models utilizing dynamic symbolic execution. Furthermore, *PyLC* [Ebr+23; Sal+23] represents a *Python*[8]-based framework that transforms ST-based and FBD-based POUs into *Python*. It simplifies test creation through integration with the *Pynguin* [LF22] test generator. *PLCverif* [DBF15] represents a user-friendly tool for model checking of ST models, and *Arcade.PLC* [BBK12] offering a comprehensive verification platform that supports static analysis and various model checking techniques across multiple industrial programming languages. Research initiatives like *PLCreX* [WS24b] further contribute to the ecosystem by enabling the simplification, transformation, analysis, and validation of IEC 61131-3 POUs, promoting reuse in model-based design and formal verification. A further

[7] `https://kframework.org/`
[8] `https://www.python.org/`

tool for static code analysis was introduced in [Pra+17]. Furthermore, the integration of modern concepts that utilize Large Language Models (LLMs) for automated test case generation was investigated in [Koz+24] and has been integrated, among others, into the *Agents4PLC* [Liu+24] framework. It combines LLM-based agents with retrieval-augmented generation and advanced prompt engineering to automate PLC code generation and verification.

In contrast to the verification within IEC 61131-3, the functional reuse of IEC 61131-3 projects within IEC 61499 projects has been explored by various transformation approaches [Sun+08; Wen+09b; Wen+09a; DV12]. Among others, specific methods for encapsulating IEC 61131-3 ST models into IEC 61499 function blocks have been developed in [WZ12]. Semantic correctness is addressed by implementing auxiliary transformations that resolve differences between the two standards in [Wen+09a]. Furthermore, synchronous semantics for IEC 61499 function blocks have been proposed by Yoong et al. [Yoo+07] that allows the application of verification techniques developed for synchronous models. Overall, these efforts demonstrate a number of approaches for reusing and transforming IEC 61131-3 models into IEC 61499 models. However, Thramboulidis and Frey [TF11] stated that "*IEC 61499 does not provide any valuable benefit in enhancing the IEC 61131 development process.*". Furthermore, according to Thramboulidis [Thr13], "*[...] 61499 cannot be considered as the effective successor of 61131 not even provide, at least with the current version, a reliable alternative for the development of industrial automation systems.* ". Cruz Salazar and Rojas Alvarado [CR14] recommended in 2014 that IEC 61499 should be applied in practice to existing processes to get accepted by the industry. In 2020, a literature review by Lyu and Brennan in the context towards IEC 61499-based distributed intelligent automation [LB21] concluded: "*[...] the first and foremost challenge for IEC 61499 industrial adoption is the widely adopted IEC 61131-3 systems, wide ranges of IEC 61131-3 software/hardware products, and proven IEC 61131-3 practices and guidelines. Research on system transformation or coexistence in the future is suggested to focus not only on the system redesign methodologies but also on the approaches to reuse or integrate existing proven practices and guidelines.*". According to the research published so far with reference to industrial applications, projects based on IEC 61131-3 still dominate industrial automation systems and will probably continue to motivate further research, especially in the context of reusability.

In addition to that, formal verification of real-time and reactive systems has significantly benefited from the adoption of synchronous programming languages [BB91]. Over the past decades, synchronous languages have evolved into a preferred technology for modeling, specifying, validating, and implementing real-time embedded applications [Ben+03]. Notable languages in this domain include the imperative synchronous language *Esterel* [BG92] and *Quartz* [SB16] that is derived from *Esterel*. *SIGNAL*, another synchronous language, emphasizes real-time system programming through a mathematical model of multiple-clocked data and event flows using an equational approach [Le +91]. *Lustre* [Hal+91] represents a data-flow oriented synchronous

language. In contrast, graphical representations such as *Statecharts* extend traditional state machines with hierarchical, concurrent, and communicative elements, offering a description language that supports modularity and comprehensibility for complex systems [Har87]. *SyncCharts* [And95] based on the synchronous paradigm and are syntactically close to *Statecharts* have further enhanced the usability and expressiveness of synchronous paradigms. Furthermore, SCCharts [Han+14] were designed for reactive systems and are based on a Sequentially Constructive Model of Computation (SCMoC) [Han+13]. In summary, synchronous languages are proven models for the formal verification and of real-time and reactive systems. Most of these languages are integrated into model-based design frameworks or engineering tools, such as *Quartz* in *Averest*, *Esterel* and *Lustre* in *SCADE*, and SCCharts in *KIELER*.

As a consequence, few works have investigated the translation of IEC 61131-3 POUs into synchronous languages. For example, Jimenez-Fraustro and Rutten [JR01] have introduced a synchronous model for ST models and FBDs based on *SIGNAL*. Werner and Schneider [WS20] have outlined the suitability of synchronous languages like *Quartz* for modeling existing POUs compared to traditional FBDs. A translation process of CFCs to *Quartz* modules for reuse purposes was introduced in [WS21]. Additionally, a translation from FBDs to SCCharts was introduced in [WS22], which enhances functional reuse and provides an alternative model view through code refactoring within the synchronous paradigm. Furthermore, the translation of ST models into synchronous *Quartz* models and models that follow the SCMoC was introduced in [WS24a].

Optimization of data-flow models, especially in terms of minimizing hardware implementation, is a broad and well researched area [Gom+09]. Furthermore, code optimization is a crucial part of software development, and various techniques have been introduced to improve the performance, resources, and readability of source code, such as simplifying Boolean logic using *Karnaugh maps* [Kar53]. One of the challenges in optimizing code for industrial data-flow models such as IEC 61131-3 FBDs is the lack of information of blocks provided by external libraries. For this reason, Werner and Schneider [WS23] have introduced an optimization process considering potentially optimizable submodels and non-modifiable components of a data-flow graph.

1.3. Outline

The thesis starts with an introduction to IEC 61131-3 POUs and model-based design using the *Averest* and *KIELER* frameworks in Chapter 2.

The formal syntax and semantics of the relevant software models are defined in Chapter 3. For this purpose, the definitions from existing descriptions and specifications of the different software models are summarized and extended by further aspects. The goal is to provide a formal specification of the individual language constructs, which will be referenced in the following chapters.

Hypothesis **H1** will be a direct result of Chapter 4, 5, 6, and 7. Specifically, Chapter 4 defines a detailed model transformation from ω_{st}^{φ} to ω_{qrz} and Chap-

ter 5 defines a detailed model transformation from ω_{st}^{φ} to ω_{scl}. Both model transformations are evaluated with appropriate ST model examples. Similarly, Chapter 6 defines a detailed model transformation from ω_{fbd}^{φ} to ω_{qrz} and Chapter 7 defines a detailed model transformation from ω_{fbd}^{φ} to ω_{scd}. These model transformations are also evaluated with appropriate examples.

The outcome of Chapter 8 will confirm hypothesis **H2**. This chapter introduces a configurable formal methods-based optimization approach of data-flow models using *NuSMV*, SMT *Z3Py*, and *PLCreX*, and analyzes the average optimization potential of real-world applications.

Finally, Chapter 9 will cover hypothesis **H3**. With the goal of synthesizing *Quartz* models ω_{qrz} into control-flow oriented SCCharts ω_{scc}, Chapter 9 introduces a pattern-based *Quartz* code refactoring intended for data-flow oriented *Quartz* models, followed by a detailed model transformation from ω_{qrz} to ω_{scc}.

In addition to the summaries in the different chapters, a short summary of the main results is given in Chapter 10.

Chapter 2

Background

Contents

This chapter introduces the main software models and frameworks used in the following chapters. The first section provides an overview of IEC 61131-3 POUs, including both textual ST models and graphical FBDs. The second section presents a model-based design approach using the *Averest* framework and the *Quartz* language. Then, a text-first design approach using the *KIELER* framework together with SCL, data-flow oriented and control-flow oriented SCCharts is presented. Finally, the chapter introduces the *PLCreX* project, which is used to evaluate the theoretical concepts in the following chapters.

2.1. IEC 61131-3 Program Organization Units

This thesis focuses on time-triggered applications, which are widely used in IEC 61131-based PLCs [Thr13] and described as reactive systems with cyclic data processing behavior [Bec+15]. In this model of computation, in each cycle a process image of the input variables is first created. Then, during the POU scan, the new values for the output variables (as determined by the POU

logic) are written to a dedicated buffer. Finally, the output values stored in this buffer are transferred by the system to the output variables [Lew98; VK02]. This work focuses on the modeling of POUs, thus neglecting the process images of the physical IEC 61131-3 system, but taking into account how instances process interfaces. POUs are organized in POUs and implemented in one of the two textual or one of the three graphical languages described in the IEC 61131-3 standard [GDV14], where this thesis focuses on textual ST models and graphical FBDs.

2.1.1. Textual Structured Text Models

The imperative sequential language ST is derived from *Pascal* and comes with constructs typical for the implementation of sequential algorithms [GDV14]. Unlike in *Pascal*, recursion is not allowed[1] in traditional ST-based POU variants *Program*, *Function*, and *Function Block* [Lew98; JT10]. A simple ST example is shown in Listing 2.1 using a rising edge detector [GDV14]. Basi-

```
1  FUNCTION_BLOCK R_TRIG
2    VAR_INPUT CLK: BOOL; END_VAR
3    VAR_OUTPUT Q: BOOL; END_VAR
4    VAR M: BOOL; END_VAR
5
6    Q := CLK AND NOT M;
7    M := CLK;
8  END_FUNCTION_BLOCK
```

Listing 2.1: *ST model of a rising edge detector*

cally, statements in ST models are executed sequentially, depending on their order within the POU. In contrast to the SMoC and SCMoC, every statement basically consumes time greater than zero, which is a crucial aspect of the transition from ST to synchronous models and will be addressed in more detail in the following chapters. The formal syntax and semantics of the relevant ST constructs in this thesis are specified in Section 3.2.

2.1.2. Graphical Function Block Diagrams

Graphical FBDs are implemented by dragging and dropping predefined or user-defined functions, function blocks, constants, or variables onto a workspace. The visual connection of these elements represents the flow of data from left to right [Lew98; GDV14], where functions and function blocks are executed according to their individual execution order identifiers. CFCs provide a less structured approach with more flexible placement and connection of blocks and variables [Lew98], and are thus interpreted in research as extended FBDs [Bia16]. Some PLC development environments imply the flexibility of CFCs in IEC 61131-3 FBDs, such as *Beremiz*[2] and *SafetyProg*[3].

[1] Recursion is possible in the context of object-oriented IEC 61131-3 features [GDV14].
[2] `https://beremiz.org/`
[3] `https://www.phoenixcontact.com/de-de/produkte/software/sps-programmierung`

For this reason, FBDs will be treated with the flexibility of CFCs in the further course of this thesis, and CFCs will simply be referred to as FBDs. A simple FBD example is shown in Figure 2.1 using the rising edge detector [GDV14]. Similar to statements in ST models, functions, function blocks, and

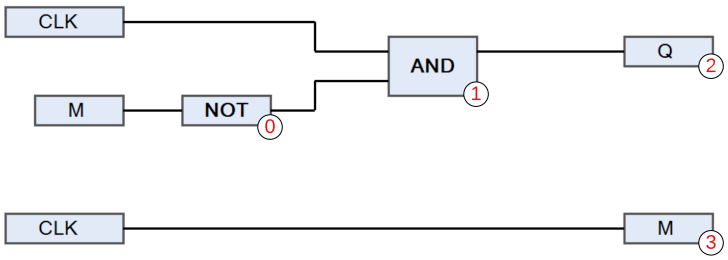


Figure 2.1.: *FBD of a rising edge detector*

assignments in FBDs are executed sequentially according to their individual execution order identifiers (such as the order labeled with red numbers in the example). Consequently, the consumption of time greater than zero is again a crucial aspect of the transition from FBDs to synchronous models and will therefore be addressed in more detail in the following chapters. The formal syntax and semantics of the relevant FBD constructs in this thesis are specified in Section 3.2.

2.2. The *Averest*-Framework

Averest is a framework for model-based design of embedded systems that includes a set of tools for simulation, synthesis, verification, synthesis, and other purposes [SS06]. In particular, it includes a compiler that translates models implemented in the imperative synchronous *Quartz* language into a symbolically represented transition system in *Averest*'s interchange format. This enables the use of *Quartz* models in a model-based design approach and formal verification. Therefore, the transformation of IEC 61131-3 POUs to *Quartz* is a key objective of this thesis.

2.2.1. Synchronous *Quartz* Models

Quartz models are organized in modules and can include user-defined functions via preprocessor directives and instantiate other *Quartz* modules [Sch09]. A simple example is shown in Listing 2.2 using the rising edge detector [GDV14].

```
1  module R_TRIG(bool ?CLK, bool !Q){
2    bool M;
3
4    Q = CLK & !M;
5    next(M) = CLK;
6    pause;
7  }
```

Listing 2.2: *Quartz model of a rising edge detector*

Quartz follows the classical SMoC [Hal98; Ben+03; Sch09], i.e., a model is partitioned into *macro steps*, which correspond to interactions between the reactive system and its environment, and *micro steps*, which are executed in zero time. The *macro steps* can be separated in *Quartz* with the special statement `pause` [Sch09], which stops the control flow at this statement. The control flow will resume at this point in the next *macro step*, assuming there is no surrounding suspension or abortion. In each *macro step*, first all input variables are read, then all output variables are computed with respect to the internal state. At the end of each *macro step*, the internal state will be updated. This is a crucial aspect of the transformations and will be addressed in more detail in the following chapters. The formal syntax and semantics of the relevant *Quartz* constructs in this thesis are specified in Section 3.3.

2.3. The *KIELER*-Framework

The *KIELER* framework is a text-first approach for the automatic diagramming of complex systems. It is designed to combine textual and diagrammatic representations by automatically synthesizing text-based models into visual diagrams [Kas+24]. *KIELER* supports multiple languages in model design and can synthesize models written in different domain-specific languages. For example, it includes SCCharts [Han+14] and the imperative SCL language [Han+13]. In the further course of this thesis, SCCharts will be distinguished between control-flow oriented and data-flow oriented [Gri+20]. As a result, another key objective of this thesis is to transform IEC 61131-3 POUs into SCL models and SCCharts.

2.3.1. Sequentially Constructive Language Models

The SCL language was introduced to illustrate the SCMoC [Han+13] and similar to *Quartz*, they are organized in modules. A simple example is shown in Listing 2.3 using the rising edge detector [GDV14]. Unlike the traditional

```
1  module R_TRIG{
2    input bool CLK;
3    output bool Q;
4    bool M;
5
6    Q = CLK & !M;
7    M = CLK;
8  }
```

Listing 2.3: *SCL model of a rising edge detector*

SMoC, the SCMoC allows variables to be read and written in the same *macro step*, as long as program sequentiality provides sufficient scheduling information to prevent race conditions [Han+13]. As a result, this language style is close to the style of traditional imperative languages such as ST, which is derived from *Pascal*. SCL programs also work in *macro steps*, where in each step first all inputs are read, and then all active (currently instantiated) threads are

executed until they either terminate or reach a `pause` statement, and at the end output variables are written to the environment [Han+13]. The formal syntax and semantics of the relevant SCL constructs in this thesis are specified in Section 3.4.

2.3.2. Control-Flow Oriented Sequentially Constructive Statecharts

SCCharts were introduced in the context of safety-critical reactive systems [Han+14] and are described in this thesis as control-flow oriented SCCharts to distinguish them from data-flow oriented SCCharts [Gri+20]. A simple example is shown in Figure 2.2 using the rising edge detector [GDV14] and both representations, the textual SCChart on the left and the resulting graph on the right. SCCharts are based on the SCMoC, which ensures deterministic

```
1  scchart R_TRIG {
2    input bool CLK
3    output bool Q
4    bool M
5
6    region:
7      initial state S0
8      immediate do Q = CLK & !
           M; M = CLK go to S1
9      final state S1
10 }
```

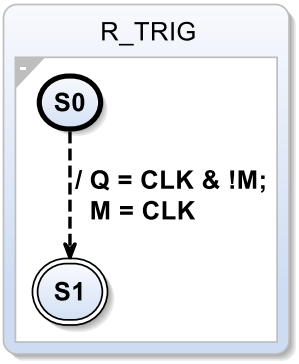


Figure 2.2.: *Control-flow oriented SCChart of a rising edge detector*

concurrency and that accesses to shared variables are sequenced to avoid conflicts despite concurrency [Han+14]. The formal syntax and semantics of the relevant control-flow oriented SCChart constructs in this thesis are specified in Section 3.5.

2.3.3. Data-Flow Oriented Sequentially Constructive Statecharts

Data-flow oriented SCCharts are an extension of the traditional control-flow oriented SCCharts and were introduced as part of a transformation approach from *Lustre* to SCCharts [Gri+20]. A simple example is shown in Figure 2.3 using the rising edge detector [GDV14], forced sequential scheduling, and both representations, the textual SCChart on the left and the resulting graph on the right. The data-flow extension is based on the SCMoC, which supports deterministic programming with sequential scheduling information [Gri+20]. Although this adaptation provides a framework where data-flow models can be seamlessly combined with control flow structures, this thesis distinguishes between control-flow oriented SCCharts and data-flow oriented SCCharts. The formal syntax and semantics of the relevant data-flow oriented SCChart constructs in this thesis are specified in Section 3.5.

```
1  scchart R_TRIG{
2    input bool CLK
3    output bool Q
4    bool M
5
6    dataflow:
7      Q = CLK & !M;
8      M = CLK;
9  }
```

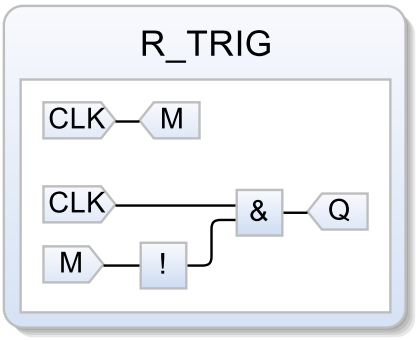


Figure 2.3.: *Data-flow oriented SCChart of a rising edge detector*

2.4. The *PLCreX* Project

PLCreX [WS24b] is a modular command line interface application tailored for IEC 61131-3 ST models, FBDs, and beyond. The project was initially motivated by research related to this thesis and is designed with a focus on issues such as verification, reuse, and reliability, among others. The high-level architecture, design principles, and interfaces to external frameworks are shown in Figure 2.4. *PLCreX* is being developed in *Python* and is intended to be used

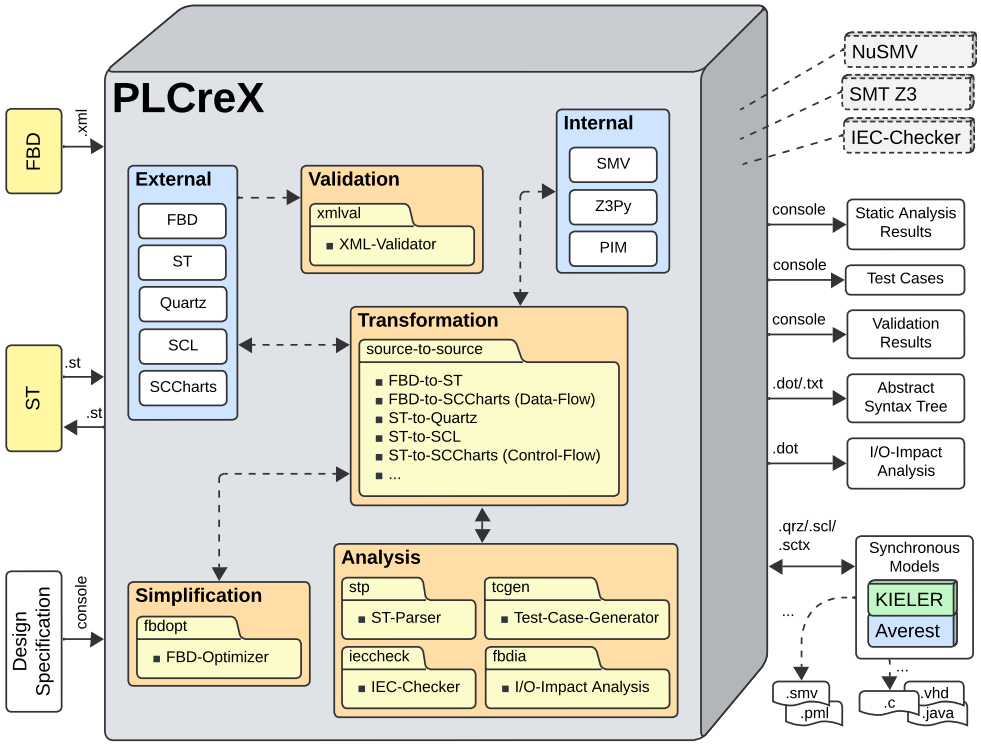


Figure 2.4.: *High-level architecture and design principles of* PLCreX *[WS24b]*

as an interface between traditional IEC 61131-3 POUs and the *Averest* and *KIELER* frameworks, or as a stand-alone analysis tool for real-world POUs

and their challenges. Although the detailed design flow depends on the individual feature, the basic concept of *PLCreX* is to process the models internally as an abstract syntax tree-like model, which is called *PLCreX* Intermediate Model (PIM) [WS24b]. The core features can be divided into four categories.

1. **Simplification:** This feature simplifies data-flow models following the optimization process introduced in Chapter 8.
2. **Transformation:** This category combines various model-to-model transformations that can be linked together.
3. **Analysis:** This category combines solutions for real-world challenges. As an example, it includes test case generation considering statement coverage, decision coverage, modified condition/decision coverage, and multiple condition coverage [Kel+01].
4. **Validation:** This feature is intended to validate POUs that are processed as *PLCopen* `xml` format [PLC09].

Chapter 3

Syntax and Formal Semantics

Contents

This chapter introduces the syntax and formal semantics of the considered software models. For this purpose, the definitions from existing descriptions and specifications of the various software models are summarized and extended with additional aspects. The goal is to provide a formal specification of the individual language constructs that will be referenced in the following chapters. The methodology is demonstrated in detail using IEC 61131-3 ST models and FBDs as examples (with isolated list items moved to the appendix). For the sake of readability, the other software models are summarized with references to the definitions in the appendix.

To this end, the first section provides an overview of the notations and statements. The second section defines the syntax and semantics of IEC 61131-3

ST models and FBDs. Then the syntax and semantics of *Quartz* are summarized. The last two sections summarize the syntax and semantics of SCL and SCCharts.

3.1. Preliminary Definitions

This section introduces the symbols and notations that will be used throughout the following chapters. These notations are essential for a complete understanding of the content of the subsequent chapters. As a first notation, the structural operational semantics (SOS) transition rules [Plo04] are defined as follows:

- f: denotes an instantaneous flag, which represents a *micro step* or current PLC cycle if true, *macro step* otherwise, but still the same PLC cycle [WS24a]
- ξ: denotes the environment of a current PLC cycle or *macro step* in synchronous models
- Σ: denotes a set of statements to be executed
- $\varphi_1 \wedge \cdots \wedge \varphi_n$: denotes assumptions
- $[\![\boxdot]\!]_\xi$: denotes the evaluation of $\boxdot$ in environment ξ
- Σ': denotes the residual set of statements for the next *micro step* or *macro step* (depending on f)
- $\mathcal{D}$: denotes the set of actions that are executed in the current step

Equation 3.1 represents SOS transition rules with assumptions and Equation 3.2 SOS transition rules that are always true, i.e., $\varphi_1 \wedge \cdots \wedge \varphi_n := \mathit{true}$.

$$\frac{\varphi_1 \wedge \cdots \wedge \varphi_n}{\langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \Sigma', \mathcal{D}, f \rangle} \tag{3.1}$$

$$\langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \Sigma', \mathcal{D}, f \rangle \tag{3.2}$$

Furthermore, the type system of expressions is defined using the following notation:

- $\pi_1 : \alpha_1 \quad \cdots \quad \pi_n : \alpha_n$: denotes the set of arguments
- τ: denotes an expression instance
- β: denotes the resulting data type

Equation 3.3 represents the full notation for type system definitions.

$$\frac{\pi_1 : \alpha_1 \quad \cdots \quad \pi_n : \alpha_n}{\tau : \beta} \tag{3.3}$$

The *PLCopen* related symbols [PLC09] are defined as follows and will be used frequently in the following chapters:

- a_n: denotes the name attribute (`name`)
- a_{iN}: denotes the instance name attribute (`instanceName`)
- a_{tN}: denotes the type name attribute (`typeName`)
- a_{eOI}: denotes the execution order identifier attribute (`executionOrderId`)
- a_{rLI}: denotes the reference local identifier attribute (`refLocalId`)
- a_{lI}: denotes the local identifier attribute (`localId`)
- a_{fP}: denotes the formal parameter attribute (`formalParameter`)
- e_e: denotes an expression element (`expression`)
- $e_{b^{inst}}$: denotes a block instance identifier (`block`)
- $e_{b^{fun'}}$: denotes a user-defined function block identifier (`block`)
- $e_{b^{fun}}$: denotes a function identifier (`block`)
- e_d: denotes a derived identifier element (`derived`)
- e_i: denotes an interface element (`interface`)
- e_{rT}: denotes a return type element (`returnType`)
- e_{lVs}: denotes a local variable element (`localVars`)
- e_{iVs}: denotes an input variable element (`inputVars`)
- e_{oVs}: denotes an output variable element (`outputVars`)
- e_{iOVs}: denotes an inout variable element (`inOutVariables`)
- e_{iV}: denotes an block input variable element (`inVariable`)
- e_{oV}: denotes an block output variable element (`outVariable`)
- e_{iOV}: denotes an block inout variable element (`inOutVariable`)

There are also generic notations used in the specifications and algorithms, which are defined as follows:

- $[\boxdot]$: denotes an optional element $\boxdot$
- `rhs(`$\boxdot$`)` denotes the right-hand side of $\boxdot$
- `lhs(`$\boxdot$`)` denotes the left-hand side of $\boxdot$
- η^i: denotes an integer expression
- η^r: denotes a floating-point expression
- λ: denotes a bit vector expression
- λ^b: denotes a boolean expression
- π, k, x, y, w: denotes a variable or compile-time constant expression
- $\mathcal{I}$: denotes the set of input variables
- $\mathcal{O}$: denotes the set of output variables

Overall, Table 3.1 provides a comprehensive overview of the considered language constructs and definitions of the various software models, where control-flow oriented SCCharts denoted as SCChart_c and data-flow oriented SCCharts denoted as SCChart_d. It is worth noting that the overview does not show all supported instruction sets of the languages, but only those (and specific instruction set combinations) that are relevant for the approaches in this thesis.

Table 3.1.: *Overview of software model syntax and semantics definitions*

Construct		**ST**	**FBD**	**Quartz**	**SCL**	**SCChart$_c$**	**SCChart$_d$**	**Description**
$\Omega_{\vartheta}^{[\varphi]}$	φ	1	1	-	-	-	-	POU variant
	δ_{ω}	1	1	1	1	1	1	declaration
$\Delta_{imports}$		-	-	1	-	-	1	imports
Δ_{idcl}	Δ_{in}	1	1	1	1	1	1	input variables
	Δ_{out}	1	1	1	1	1	1	output vars.
	Δ_{inout}	1	1	1	1	1	1	inout variables
Δ_{vdcl}	Δ_{local}	1	1	1	1	1	1	local variables
	Δ_{inst}	1	1	-	-	1	1	instance vars.
$\mathcal{A}$	$\mathcal{A}_{bv}$	1	1	1	1	1	1	bit vector vars.
	$\mathcal{A}_{dur}$	1	1	1	1	1	1	duration vars.
	$\mathcal{A}_{i}$	1	1	1	1	1	1	integer vars.
	$\mathcal{A}_{r}$	1	1	1	1	1	1	float variables
	$\mathcal{A}^{+}$	1	1	1	1	1	1	data type fields
$\mathcal{T}$	$\mathcal{T}_{arith}$	1-3	1-3	1-3	1-3	1-3	1-3	arithmetic expr.
	$\mathcal{T}_{bv}$	1-3	1-3	1-3	1-3	1-3	1-3	bitwise expr.
	$\mathcal{T}_{comp}$	1-3	1-3	1-3	1-3	1-3	1-3	comparison expr.
	$\mathcal{T}_{cond}$	1-3	1-3	1-3	1-3	1-3	1-3	conditional expr.
	$\mathcal{T}_{misc}$	1-3	1-3	1-3	1-3	1-3	1-3	misc. expr.
Σ	Σ_{abort}	-	-	2	-	2	-	abortions
	Σ_{ass}	2	2	2	2	2	2	assignments
	Σ_{await}	-	-	2	-	2	-	wait stats.
	Σ_{conc}	-	-	2	-	2	2	parallel stats.
	Σ_{cond}	2	-	2	2	2	-	conditions
	Σ_{halt}	-	-	2	-	2	-	halt stats.
	Σ_{inv}	2	2	2	-	-	2	invocations
	Σ_{loop}	2	-	2	2	2	-	loops
	$\Sigma_{nothing}$	-	-	2	-	2	-	nothing stats.
	Σ_{pause}	-	-	2	2	2	-	pause stats.
	Σ_{seq}	2	2	2	2	2	2	sequences

Notes: syntax and semantics defined (1), syntax and SOS transition rules defined (2), type system defined (3), no definitions (-)

3.2. IEC 61131-3 FBDs and ST Models

This section provides an overview of the syntax and semantics of an FBD $\omega^{\varphi}_{fbd} \in \Omega^{\varphi}_{fbd}$ and an ST model $\omega^{\varphi}_{st} \in \Omega^{\varphi}_{st}$, where $\omega^{\varphi}_{pou} \in \{\omega^{\varphi}_{fbd}, \omega^{\varphi}_{st}\}$. The definitions of syntax and semantics are based on IEC 61131-3 [GDV14] and a transformation from ST to synchronous models [WS24a]. It is worth noting that the definitions are limited to the constructs that are relevant in the following chapters.

3.2.1. POU Variants and Declaration

There are different POU variants φ, where this thesis considers the variants *program* ($\varphi = prg$), *function block* ($\varphi = fb$), and *function* ($\varphi = fun$), which are defined as follows:

Definition 3.1 (Syntax of POU elements). *The syntax of ω^{φ}_{pou} is declared as follows, assuming fixed order of $\Delta_{idcl}(\omega^{\varphi}_{pou})$, $\Delta_{vdcl}(\omega^{\varphi}_{pou})$, and $\Sigma(\omega^{\varphi}_{pou})$ (see Appendix A.1.1 for the full list):*

- $\delta_{\omega}(\omega^{fb}_{pou}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{FUNCTION\_BLOCK } a_n(\omega^{fb}_{pou}) \\ \quad [\Delta_{idcl}(\omega^{fb}_{pou})] \quad [\Delta_{vdcl}(\omega^{fb}_{pou})] \quad [\Sigma(\omega^{fb}_{pou})] \\ \texttt{END\_FUNCTION\_BLOCK} \end{array} \right\}$

Definition 3.2 (Semantics of POU elements). *The semantics of ω^{φ}_{pou} are defined as follows (see Appendix A.1.1 for the full list):*

- $[\![\delta_{\omega}(\omega^{fb}_{pou})]\!]_{\xi} \stackrel{def}{=} \{$ *defines a POU element ω^{fb}_{pou} with $\Sigma(\omega^{fb}_{pou})$, started at time t with initial conditions set by $\Delta_{idcl}(\omega^{fb}_{pou})$ and $\Delta_{vdcl}(\omega^{fb}_{pou})$ when invoked, terminating at time $t + \theta$ and preserving internal state across invocations by another POU element, i.e., ω^{fb}_{pou} has memory* $\}$

3.2.2. POU Interfaces

POUs may be equipped with interfaces $\Delta_{idcl}(\omega^{\varphi}_{pou})$, where this thesis considers *input* variables $\Delta_{in}(\omega^{\varphi}_{pou})$, *output* variables $\Delta_{out}(\omega^{\varphi}_{pou})$, and *inout* variables $\Delta_{inout}(\omega^{\varphi}_{pou})$, which are defined as follows:

Definition 3.3 (Syntax of POU interfaces). *The syntax of $\Delta_{in}(\omega^{\varphi}_{pou})$, $\Delta_{out}(\omega^{\varphi}_{pou})$, and $\Delta_{inout}(\omega^{\varphi}_{pou})$ is defined as follows (see Appendix A.1.2 for the full list):*

- $\Delta_{out}(\omega^{\varphi}_{pou}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{VAR\_OUTPUT} \\ \quad x_1 \texttt{ : } \alpha_1^{[+]}\texttt{[:=}w_1\texttt{];} \\ \quad \vdots \\ \quad x_n \texttt{ : } \alpha_n^{[+]}\texttt{[:=}w_n\texttt{];} \\ \texttt{END\_VAR} \end{array} \right\}$

Definition 3.4 (Semantics of POU interfaces). *The semantics of* $\Delta_{in}(\omega^{\varphi}_{pou})$, $\Delta_{out}(\omega^{\varphi}_{pou})$, *and* $\Delta_{inout}(\omega^{\varphi}_{pou})$ *are defined as follows, noting that field elements can only be used as input variables, output variables, and inout variables of function elements, and as inout variables of function block elements (see Appendix A.1.2 for the full list):*

- $[\![\Delta_{out}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{def}{=} \{$ *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_{\xi}, \ldots, [\![\alpha_n^{[+]}]\!]_{\xi}$. *These variables are assigned to optional predefined default values* $[\![w_1]\!]_{\xi}, \ldots, [\![w_n]\!]_{\xi}$ *at time* t *when* ω^{φ}_{pou} *is invoked, and passed to invoking element with final computed values when* ω^{φ}_{pou} *terminates at time* $t+\theta$, *i.e., all instructions of* ω^{φ}_{pou} *have been processed in the current PLC cycle.* $\}$

3.2.3. Local Variables in POUs

POUs may contain local variables $\Delta_{vdcl}(\omega^{\varphi}_{pou})$, where variables are distinguished between standard variables $\Delta_{local}(\omega^{\varphi}_{pou})$ and instance variables $\Delta_{inst}(\omega^{\varphi}_{pou})$, which are defined as follows:

Definition 3.5 (Syntax of local variables in POUs). *The syntax of* $\Delta_{local}(\omega^{\varphi}_{pou})$ *and* $\Delta_{inst}(\omega^{\varphi}_{pou})$ *is defined as follows (see Appendix A.1.3 for the full list):*

- $\Delta_{local}(\omega^{\varphi}_{pou}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{VAR} \\ \quad x_1 \ : \ \alpha_1^{[+]} \texttt{[:=}w_1\texttt{]}\texttt{;} \\ \quad \vdots \\ \quad x_n \ : \ \alpha_n^{[+]} \texttt{[:=}w_n\texttt{]}\texttt{;} \\ \texttt{END\_VAR} \end{array} \right\}$

Definition 3.6 (Semantics of local variables in POUs). *The semantics of* $\Delta_{local}(\omega^{\varphi}_{pou})$ *and* $\Delta_{inst}(\omega^{\varphi}_{pou})$ *are defined as follows (see Appendix A.1.3 for the full list):*

- $[\![\Delta_{local}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{def}{=} \{$ *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_{\xi}, \ldots, [\![\alpha_n^{[+]}]\!]_{\xi}$. *Depending on the POU variant, the variables keep their values from previous invocation or are assigned to optional predefined default values* $[\![w_1]\!]_{\xi}, \ldots, [\![w_n]\!]_{\xi}$ *at time* t *when* ω^{φ}_{pou} *is invoked. These variables can be modified and processed locally until* ω^{φ}_{pou} *terminates at time* $t+\theta$, *i.e., all instructions of* ω^{φ}_{pou} *have been processed in the current PLC cycle.* $\}$

3.2.4. Elementary IEC 61131-3 Data Types and Fields

IEC 61131-3 POUs are capable of processing a variety of elementary data types $\mathcal{A}(\omega^{\varphi}_{pou})$ and fields $\mathcal{A}^{+}(\omega^{\varphi}_{pou})$, where this thesis considers *bit vector data types* $\mathcal{A}_{bv}(\omega^{\varphi}_{pou})$, *integer data types* $\mathcal{A}_{i}(\omega^{\varphi}_{pou})$, *floating-point data types* $\mathcal{A}_{r}(\omega^{\varphi}_{pou})$,

numeric data types $\mathcal{A}_i(\omega_{pou}^{\varphi}) \cup \mathcal{A}_r(\omega_{pou}^{\varphi})$, and *duration data types* $\mathcal{A}_{dur}(\omega_{pou}^{\varphi})$. Furthermore, it is assumed, that array indices start at index `0` by default. $\mathcal{A}(\omega_{pou}^{\varphi})$ and $\mathcal{A}^{+}(\omega_{pou}^{\varphi})$ are defined as follows, where the semantics are defined as a tuple $\langle$`<MIN>`, `<MAX>`, `<DEFAULT>`$\rangle$:

Definition 3.7 (Syntax of elementary IEC 61131-3 data types and fields). *The syntax of* $\alpha(\omega_{pou}^{\varphi}) \in \mathcal{A}(\omega_{pou}^{\varphi})$ *and* $\alpha^{+}(\omega_{pou}^{\varphi}) \in \mathcal{A}^{+}(\omega_{pou}^{\varphi})$ *is defined as follows (see Appendix A.1.4 for the full list):*

Syntax of bit vector data types $\alpha_{bv}(\omega_{pou}^{\varphi}) \in \mathcal{A}_{bv}(\omega_{pou}^{\varphi})$*:*

- $\alpha_{bv}^{bool}(\omega_{pou}^{\varphi}) \stackrel{def}{=} \{\texttt{BOOL}\}$ *denotes boolean values*
- $\alpha_{bv}^{byte}(\omega_{pou}^{\varphi}) \stackrel{def}{=} \{\texttt{BYTE}\}$ *denotes single byte bit masks*
- $\alpha_{bv}^{word}(\omega_{pou}^{\varphi}) \stackrel{def}{=} \{\texttt{WORD}\}$ *denotes two byte bit masks*

Definition 3.8 (Semantics of elementary IEC 61131-3 data types and fields). *In accordance with IEC 61131-3 and assuming the limitation of the duration data type* $\alpha_{dur}^{time}(\omega_{pou}^{\varphi})$ *to* `32 Bit`*, the semantics of* $\alpha(\omega_{pou}^{\varphi}) \in \mathcal{A}(\omega_{pou}^{\varphi})$ *and* $\alpha^{+}(\omega_{pou}^{\varphi}) \in \mathcal{A}^{+}(\omega_{pou}^{\varphi})$ *are defined as follows (see Appendix A.1.4 for the full list):*

Semantics of bit vector data types $\alpha_{bv}(\omega_{pou}^{\varphi}) \in \mathcal{A}_{bv}(\omega_{pou}^{\varphi})$*:*

- $[\![\alpha_{bv}^{bool}(\omega_{pou}^{\varphi})]\!]_{\xi} \stackrel{def}{=} \langle \{false, true\}, false \rangle$
- $[\![\alpha_{bv}^{byte}(\omega_{pou}^{\varphi})]\!]_{\xi} \stackrel{def}{=} \langle \{\texttt{00}_{hex}, \texttt{ff}_{hex}\}, \texttt{00}_{hex} \rangle$
- $[\![\alpha_{bv}^{word}(\omega_{pou}^{\varphi})]\!]_{\xi} \stackrel{def}{=} \langle \{\texttt{0000}_{hex}, \texttt{ffff}_{hex}\}, \texttt{0000}_{hex} \rangle$

3.2.5. Expressions in POUs

IEC 61131-3 POUs are capable of processing a wide range of expressions $\mathcal{T}(\omega_{pou}^{\varphi})$, where this thesis considers *constants and general expressions* $\mathcal{T}_{misc}(\omega_{pou}^{\varphi})$, *comparison operators* $\mathcal{T}_{comp}(\omega_{pou}^{\varphi})$, *arithmetic operators* $\mathcal{T}_{arith}(\omega_{pou}^{\varphi})$, *bitwise operators* $\mathcal{T}_{bv}(\omega_{pou}^{\varphi})$, and *conditional operators* $\mathcal{T}_{cond}(\omega_{pou}^{\varphi})$, which are defined as follows[1]:

Definition 3.9 (Syntax of expressions in POUs). *The syntax of* $\mathcal{T}_{misc}(\omega_{pou}^{\varphi})$*,* $\mathcal{T}_{comp}(\omega_{pou}^{\varphi})$*,* $\mathcal{T}_{arith}(\omega_{pou}^{\varphi})$*,* $\mathcal{T}_{bv}(\omega_{pou}^{\varphi})$*, and* $\mathcal{T}_{cond}(\omega_{pou}^{\varphi})$ *is defined as follows (see Appendix A.1.5 for the full list):*

Syntax of comparison operators $\tau_{comp}(\omega_{pou}^{\varphi}) \in \mathcal{T}_{comp}(\omega_{pou}^{\varphi})$*:*

- $\tau_{comp}^{eq}(\omega_{pou}^{\varphi}) \stackrel{def}{=} \begin{cases} \{\pi_1 \texttt{ = } \pi_2\}, & \text{if } \varphi = st \\ \{\texttt{EQ(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ *denotes equality*

[1] For ST models, it is possible to use the operators as equivalent function calls [GDV14].

- $\tau^{ne}_{comp}(\omega^{\varphi}_{pou}) \stackrel{def}{=} \begin{cases} \{\pi_1 \texttt{ <> } \pi_2\}, & \text{if } \varphi = st \\ \{\texttt{NE(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ *denotes inequality*

Definition 3.10 (Type system of expressions in POUs). *The type system of $\tau(\omega^{\varphi}_{pou})$ is defined as follows (see Appendix A.1.5 for the full list):*

Type system of comparison operators $\tau^{\gamma}_{comp}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{comp}(\omega^{\varphi}_{pou})$, where $\gamma \in \{eq, ne, gt, ge, lt, le\}$:

- $$\frac{\pi_1 : \alpha(\omega^{\varphi}_{pou}) \quad \pi_2 : \alpha(\omega^{\varphi}_{pou})}{\tau^{\gamma}_{comp}(\omega^{\varphi}_{pou}) : \alpha^{bool}_{bv}(\omega^{\varphi}_{pou})}$$

Definition 3.11 (Semantics of expressions in POUs). *The semantics of $\tau(\omega^{\varphi}_{pou}) \in \mathcal{T}(\omega^{\varphi}_{pou})$ are defined as follows (see Appendix A.1.5 for the full list):*

Semantics of comparison operators $\tau_{comp}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{comp}(\omega^{\varphi}_{pou})$:

- $[\![\tau^{eq}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} = [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{ne}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} \neq [\![\pi_2]\!]_{\xi}$

Definition 3.12 (SOS transition rules of expressions in POUs). *The SOS transition rules of $\tau(\omega^{\varphi}_{pou}) \in \mathcal{T}(\omega^{\varphi}_{pou})$ are defined as follows:*

- $\langle \xi, \tau(\omega^{\varphi}_{pou}) \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{[\![\tau(\omega^{\varphi}_{pou})]\!]_{\xi}\}, true \rangle$

3.2.6. POU Invocations in POUs

It is possible for POUs to invoke user-defined *functions* $\Sigma^{f'}_{inv}(\omega^{\varphi}_{pou})$ and *function blocks* $\Sigma^{fb}_{inv}(\omega^{\varphi}_{pou})$. During the execution of these invoked models, the execution of the invoking POU is paused and resumed when the instance terminates. As mentioned in Section 3.2.1, only *function blocks* have memory. However, both *functions* and *function blocks* terminate within the same PLC cycle.

Definition 3.13 (Syntax of POU invocations in POUs). *The syntax of $\sigma^{f'}_{inv}(\omega^{\varphi}_{pou}) \in \Sigma^{f'}_{inv}(\omega^{\varphi}_{pou})$ and $\sigma^{fb}_{inv}(\omega^{\varphi}_{pou}) \in \Sigma^{fb}_{inv}(\omega^{\varphi}_{pou})$ is defined as follows, where elements of graphical FBDs can be derived from* PLCopen *export:*

- $\sigma^{f'}_{inv}(\omega^{\varphi}_{pou}) \stackrel{def}{=} \left\{ k\texttt{(}\mathcal{I}(\sigma^{f'}_{inv}(\omega^{\varphi}_{pou}))\texttt{,}\mathcal{O}(\sigma^{f'}_{inv}(\omega^{\varphi}_{pou}))\texttt{)[;]} \right\}$
- $\sigma^{fb}_{inv}(\omega^{\varphi}_{pou}) \stackrel{def}{=} \left\{ k\texttt{(}\mathcal{I}(\sigma^{fb}_{inv}(\omega^{\varphi}_{pou}))\texttt{)[;]} \right\}$

Definition 3.14 (SOS transition rules of POU invocations in POUs). *The SOS transition rules of $\sigma^{f'}_{inv}(\omega^{\varphi}_{pou}) \in \Sigma^{f'}_{inv}(\omega^{\varphi}_{pou})$ and $\sigma^{fb}_{inv}(\omega^{\varphi}_{pou}) \in \Sigma^{fb}_{inv}(\omega^{\varphi}_{pou})$ are defined as follows:*

- $\langle \xi, \sigma_{inv} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{[\![k]\!]_{\xi}\}, true \rangle$

3.2.7. Assignments in POUs

POUs can be classified into two categories with regard to their assignments. In this thesis, they are called (1) *immediate assignments* $\Sigma_{ass}^{imm}(\omega_{pou}^{\varphi})$, where the evaluation of the expressions is independent of the variable which shall be assigned, and (2) *delayed assignments* $\Sigma_{ass}^{del}(\omega_{pou}^{\varphi})$ if not. For this reason, a sequential dependency between evaluation of the expressions and assignment must be considered.

Definition 3.15 (Syntax of assignments in POUs). *The syntax of* $\sigma_{ass}^{imm}(\omega_{pou}^{\varphi}) \in \Sigma_{ass}^{imm}(\omega_{pou}^{\varphi})$ *and* $\sigma_{ass}^{del}(\omega_{pou}^{\varphi}) \in \Sigma_{ass}^{del}(\omega_{pou}^{\varphi})$ *is defined as follows, where assignments in graphical FBDs can be derived from* PLCopen *export:*

- $\sigma_{ass}^{imm}(\omega_{pou}^{\varphi}) \stackrel{def}{=} \{\ x := \tau;\ \}$
- $\sigma_{ass}^{del}(\omega_{pou}^{\varphi}) \stackrel{def}{=} \{\ x := \tau(x);\ \}$

Definition 3.16 (SOS transition rules of assignments in POUs). *The SOS transition rules of* $\sigma_{ass}^{imm}(\omega_{pou}^{\varphi}) \in \Sigma_{ass}^{imm}(\omega_{pou}^{\varphi})$ *and* $\sigma_{ass}^{del}(\omega_{pou}^{\varphi}) \in \Sigma_{ass}^{del}(\omega_{pou}^{\varphi})$ *are defined as follows:*

- $\langle \xi, \sigma_{ass}^{imm} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{x = [\![\tau]\!]_{\xi}\}, true \rangle$
- $\langle \xi, \sigma_{ass}^{del} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{x = [\![\tau(x)]\!]_{\xi}\}, true \rangle$

3.2.8. Conditions in ST Models

ST models are capable of processing different variants of condition statements, where this thesis considers *if-then* conditions $\Sigma_{cond}^{it}(\omega_{st}^{\varphi})$ and *if-then-else* conditions $\Sigma_{cond}^{ite}(\omega_{st}^{\varphi})$.

Definition 3.17 (Syntax of conditions in ST models). *The syntax of* $\sigma_{cond}^{it}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{it}(\omega_{st}^{\varphi})$ *and* $\sigma_{cond}^{ite}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{ite}(\omega_{st}^{\varphi})$ *is defined as follows (see Appendix A.1.6 for the full list):*

- $\sigma_{cond}^{ite}(\omega_{st}^{\varphi}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{IF}\ \lambda^{b}(\sigma_{cond}^{ite}(\omega_{st}^{\varphi}))\ \texttt{THEN}\ \Sigma_{1}(\sigma_{cond}^{ite}(\omega_{st}^{\varphi})) \\ \texttt{ELSE}\ \Sigma_{2}(\sigma_{cond}^{ite}(\omega_{st}^{\varphi}))\ \texttt{END\_IF} \end{array} \right\}$

Definition 3.18 (SOS transition rules of conditions in ST models). *The SOS transition rules of* $\sigma_{cond}^{it}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{it}(\omega_{st}^{\varphi})$ *and* $\sigma_{cond}^{ite}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{ite}(\omega_{st}^{\varphi})$ *are defined as follows (see Appendix A.1.6 for the full list):*

SOS transition rules of $\sigma_{cond}^{ite}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{ite}(\omega_{st}^{\varphi})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{IF } \lambda^b \texttt{ THEN } \Sigma_1 \\ \texttt{ELSE } \Sigma_2 \texttt{ END\_IF} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_2, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{IF } \lambda^b \texttt{ THEN } \Sigma_1 \\ \texttt{ELSE } \Sigma_2 \texttt{ END\_IF} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_2, true \rangle}$$

3.2.9. Loops in ST Models

ST models are capable of processing different variants of loop statements, where this thesis considers *bounded foot-controlled* loops $\Sigma^{foot}_{loop}(\omega^{\varphi}_{st})$ and *bounded head-controlled* loops $\Sigma^{head}_{loop}(\omega^{\varphi}_{st})$.

Definition 3.19 (Syntax of loops in ST models). *The syntax of* $\sigma^{foot}_{loop}(\omega^{\varphi}_{st}) \in \Sigma^{foot}_{loop}(\omega^{\varphi}_{st})$ *and* $\sigma^{head}_{loop}(\omega^{\varphi}_{st}) \in \Sigma^{head}_{loop}(\omega^{\varphi}_{st})$ *is defined as follows (see Appendix A.1.7 for the full list):*

- $\sigma^{head}_{loop}(\omega^{\varphi}_{st}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b(\sigma^{head}_{loop}(\omega^{\varphi}_{st})) \texttt{ DO} \\ \Sigma(\sigma^{head}_{loop}(\omega^{\varphi}_{st})) \texttt{ END\_WHILE} \end{array} \right\}$

Definition 3.20 (SOS transition rules of loops in ST models). *The SOS transition rules of* $\sigma^{foot}_{loop}(\omega^{\varphi}_{st}) \in \Sigma^{foot}_{loop}(\omega^{\varphi}_{st})$ *and* $\sigma^{head}_{loop}(\omega^{\varphi}_{st}) \in \Sigma^{head}_{loop}(\omega^{\varphi}_{st})$ *are defined as follows (see Appendix A.1.7 for the full list):*

SOS transition rules of $\sigma^{head}_{loop}(\omega^{\varphi}_{st}) \in \Sigma^{head}_{loop}(\omega^{\varphi}_{st})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b \texttt{ DO} \\ \Sigma \texttt{ END\_WHILE} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\mathcal{D}; \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b \texttt{ DO} \\ \mathcal{D} \texttt{ END\_WHILE} \end{array} \right\}, true \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false}{\langle \xi, \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b \texttt{ DO} \\ \Sigma \texttt{ END\_WHILE} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}$$

3.2.10. Sequences in POUs

A set of the aforementioned statements is called a sequence $\Sigma_{seq}(\omega^{\varphi}_{pou})$. By default, the statements are executed sequentially, and the sequence terminates when the last statement is terminated. This is true for both ST models and FBDs.

Definition 3.21 (Syntax of sequences in POUs). *The syntax of a sequence* $\Sigma_{seq}(\omega^{\varphi}_{pou})$ *is defined as follows, where sequences in graphical FBDs can be derived from* PLCopen *export:*

- $\Sigma_{seq}(\omega_{pou}^{\varphi}) \stackrel{def}{=} \left\{ \begin{array}{l} \sigma_1(\omega_{pou}^{\varphi}); \\ \sigma_2(\omega_{pou}^{\varphi}); \\ \vdots \\ \sigma_n(\omega_{pou}^{\varphi}); \end{array} \right\}$

Definition 3.22 (SOS transition rules of sequences in POUs). *The SOS transition rules of a sequence $\Sigma_{seq}(\omega_{pou}^{\varphi})$ are defined as follows, where ξ' represents the updated environment:*

- $$\frac{\langle \xi, \sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle \wedge \langle \xi', \sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_2, true \rangle}{\langle \xi, \{\sigma_1; \sigma_2;\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\mathcal{D}_1; \mathcal{D}_2\}, true \rangle}$$

3.3. *Quartz* Models

The following summary of syntax and semantics of a *Quartz* model $\omega_{qrz} \in \Omega_{qrz}$ is based on [Sch09; SB16; WS24a] and it is worth noting that the definitions are limited to the constructs that are relevant in the following chapters.

There is one variant of *Quartz* model elements called *module* (see Appendix A.2.1), which can be imported and instantiated by other *Quartz* modules. The imported *Quartz* models are grouped as a set $\Delta_{imports}(\omega_{qrz})$ (see Appendix A.2.2).

Furthermore, *Quartz* models may be equipped with interfaces $\Delta_{idcl}(\omega_{qrz})$. More specifically, these are *input* variables $\Delta_{in}(\omega_{qrz})$, *output* variables $\Delta_{out}(\omega_{qrz})$, and *inout* variables $\Delta_{inout}(\omega_{qrz})$. There are two categories of storage classes: (1) *event* variables are reset to their default values when they are not assigned in the current *macro step*, and (2) *memorized* variables retain the values of the previous *macro step*, unless assigned to new values in the current *macro step* (see Appendix A.2.3) [Sch09]. *Quartz* models may also contain local variables $\Delta_{vdcl}(\omega_{qrz})$ (see Appendix A.2.4).

Quartz models are capable of processing a variety of elementary data types $\mathcal{A}(\omega_{qrz})$ and fields $\mathcal{A}^{+}(\omega_{qrz})$, where this thesis considers the variants introduced in Section 3.2.4 (see Appendix A.2.5).

In addition, *Quartz* models are capable of processing a wide range of expressions $\mathcal{T}(\omega_{qrz})$, where this thesis considers the variants introduced in Section 3.2.5 (see Appendix A.2.6).

Quartz comes with different variants of *abort* statements $\Sigma_{abort}(\omega_{qrz})$, where this thesis considers *regular abort* statements $\Sigma_{abort}^{reg}(\omega_{qrz})$ and *immediate abort* statements $\Sigma_{abort}^{imm}(\omega_{qrz})$ (see Appendix A.2.7). In general, these statements are used to evaluate a condition in each step, which controls the execution of inner statements.

Moreover, *Quartz* models can be classified into two principal categories with regard to their assignments $\Sigma_{ass}(\omega_{qrz})$, where this thesis considers the variants introduced in Section 3.2.7. Consequently, it is necessary to consider the sequential dependency between evaluation and assignment. Due to the SMoC, an assignment to the left-hand side using *delayed assignments* is only permitted in the subsequent *macro step* (see Appendix A.2.8).

There are statements in *Quartz* models where sequential execution stops until a Boolean condition becomes true. These statements are *await* statements $\Sigma_{await}(\omega_{qrz})$, where this thesis considers *regular await* statements $\Sigma^{reg}_{await}(\omega_{qrz})$ and *immediate await* statements $\Sigma^{imm}_{await}(\omega_{qrz})$ (see Appendix A.2.9).

In addition to *Quartz* sequences, there are statements for parallel execution, where this thesis considers *synchronous concurrency* $\Sigma_{conc}(\omega_{qrz})$ (see Appendix A.2.10). It is worth noting that as long as the statements do not terminate, they execute their *macro steps* synchronously in lockstep, i.e., statements may interact during concurrent execution.

Quartz models are capable of processing different variants of conditions $\Sigma_{cond}(\omega_{qrz})$, where this thesis considers the variants introduced in Section 3.2.8 (see Appendix A.2.11).

There are a statements in *Quartz* models to stop the sequential execution, which are called *halt* statements $\Sigma_{halt}(\omega_{qrz})$ (see Appendix A.2.12).

It is possible in *Quartz* models to invoke other *Quartz* models through the use of statements $\Sigma_{inv}(\omega_{qrz})$ (see Appendix A.2.13). It is worth noting that there is a significant dependency on whether the invoked instances are executed sequentially or in parallel, because due to the SMoC, statements may interact during concurrent execution.

Quartz models are capable of processing different variants of loops $\Sigma_{loop}(\omega^{\varphi}_{st})$, where this thesis considers the variants introduced in Section 3.2.9 (see Appendix A.2.14). It is worth noting that body statement must not be instantaneous.

There are *nothing* statements in *Quartz* models $\Sigma_{nothing}(\omega_{qrz})$ (see Appendix A.2.15) that are introduced at this point for isolated transformations in the following chapters.

The consumption of time within *Quartz* models can be explicitly programmed with statements called `pause` $\Sigma_{pause}(\omega_{qrz})$ (see Appendix A.2.16). In general, the execution of a `pause` statement consumes one logical unit of time and separates two *macro steps*.

Finally, a set of *Quartz* statements is called a sequence $\Sigma_{seq}(\omega_{qrz})$ (see Appendix A.2.17). It is worth noting that the statements are executed in a sequential manner considering the SMoC, where the sequence terminates upon the termination of the final statement, i.e., the sequence is instantaneous if all statements are instantaneous.

3.4. SCL Models

The following summary of syntax and semantics of an SCL model $\omega_{scl} \in \Omega_{scl}$ is based on [Han+13; Han+14; WS24a] and manual studies of the latest standalone version of the *KIELER* project. It is worth noting that the definitions are limited to the constructs that are relevant in the following chapters.

Like in *Quartz* models, there is one variant of SCL model elements called *module* (see Appendix A.3.1). These models are not intended to be imported and instantiated by other SCL models due to their minimal instruction set, but can be synthesized into SCCharts by *KIELER* and then reused [WS24a].

SCL models can also be equipped with interfaces $\Delta_{idcl}(\omega_{scl})$. Possible interfaces are *input* variables $\Delta_{in}(\omega_{scl})$, *output* variables $\Delta_{out}(\omega_{scl})$, and *inout* variables $\Delta_{inout}(\omega_{scl})$. There are also different kinds of storage classes [Han+13; Sch+18], where this thesis focuses on variables that retain the values of the previous *macro step* unless they are assigned new values in the current *macro step* (as *memorized* variables in *Quartz*) to ensure consistency between the different models, although this requires an additional reset if they are intended to behave like *event* variables in *Quartz* (see Appendix A.3.2). However, Chapter 9 considers *signals* [Sch+18] to mimic event-driven execution without the need to reset interfaces, as this approach is restricted to control-flow oriented SCCharts (see Appendix A.3.2).

Additionally, SCL models may contain local variables $\Delta_{vdcl}(\omega_{scl})$ (see Appendix A.3.3).

Furthermore, SCL models are also capable of processing a variety of elementary data types $\mathcal{A}(\omega_{scl})$ and fields $\mathcal{A}^{+}(\omega_{scl})$, where this thesis considers the variants introduced in Section 3.2.4 (see Appendix A.3.4).

Moreover, SCL models are capable of processing a wide range of expressions $\mathcal{T}(\omega_{scl})$, where this thesis considers the variants introduced in Section 3.2.5 (see Appendix A.3.5).

Like *Quartz* models, SCL models can be classified into two principal categories with regard to their assignments $\Sigma_{ass}(\omega_{scl})$, where this thesis considers the variants introduced in Section 3.2.7. Due to the SCMoC, in both variants left-hand side is updated instantaneous, because sequential order is considered by default (see Appendix A.3.6).

SCL models are capable of processing different variants of conditions $\Sigma_{cond}(\omega_{scl})$, where this thesis considers the variants introduced in Section 3.2.8 (see Appendix A.3.7).

Additionally, SCL models are capable of processing different variants of loops $\Sigma_{loop}(\omega_{scl})$, where this thesis considers the variants introduced in Section 3.2.9. Due to the limited instruction set of SCL models, loop constructs are defined using a combination of a `goto` statement and a condition (see Appendix A.3.8) [WS24a]. It is worth noting that body statements of loops in SCL models are assumed not to be instantaneous by default.

A set of SCL statements is called a sequence $\Sigma_{seq}(\omega_{scl})$ (see Appendix A.3.10). It is worth noting that the statements in the SCMoC are executed in a sequential manner.

3.5. SCCharts

The following summary of syntax and semantics of a control-flow oriented SCCharts $\omega_{scc} \in \Omega_{scc}$ and a data-flow oriented SCCharts $\omega_{scd} \in \Omega_{scd}$ is based on [Han+14; Gri+20; WS23], syntax specifications on the *KIELER* homepage[2], and manual studies of the latest standalone version of the *KIELER* project. It is worth noting that the definitions are limited to the constructs that are

[2] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Syntax`

relevant in the following chapters.

This thesis distinguishes between both variants Ω_{scc} (see Appendix A.5.1) and Ω_{scd} (see Appendix A.4.1) and considers that data-flow oriented SCCharts are able to import data-flow and control-flow oriented SCCharts. The SCCharts imported by a data-flow oriented SCChart are grouped as a set $\Delta_{imports}(\omega_{scd})$ (see Appendix A.2.2).

Local variables of both control-flow oriented and data-flow oriented SCCharts are defined equivalent to those of SCL models (see Section 3.4), with those of data-flow oriented SCCharts are extended by instances of imported SCCharts (see Appendix A.4.2).

Defined interfaces, elementary data types and fields, and assignments almost equivalent to those of SCL models (see Section 3.4). The difference is that how these constructs are placed within the models. In particular, unlike in SCL models, declarations in SCCharts don't end with a semicolon. Furthermore, synthesizing data-flow oriented SCCharts to control-flow oriented SCCharts using *KIELER* shows that placing a semicolon at the end of assignments enforces sequential execution of synchronous concurrent threads, which is assumed in the further course of this thesis. This replaces the `seq` statement introduced in related work using data-flow oriented SCCharts [WS23]. Additionally, assignments in control-flow oriented SCCharts are indicated by the prefix `do`.

Similar to *await* statements in *Quartz*, there are *await* transitions in control-flow oriented SCCharts $\Sigma_{await}(\omega_{scc})$, considering *regular await* transitions $\Sigma^{reg}_{await}(\omega_{scc})$ and *immediate await* transitions $\Sigma^{imm}_{await}(\omega_{scc})$ (See Appendix A.2.9).

Each assignment of a data-flow oriented SCCharts is basically executed in a separate thread concurrently to the others [Gri+20; WS22]. Although sequential dependency is enforced as mentioned earlier, this will be interpreted as *synchronous concurrency* $\Sigma_{conc}(\omega_{scd})$ (see Appendix A.4.4) in the further course of this thesis. In contrast, *regions* in control-flow oriented SCCharts are also being executed concurrently and are interpreted as *synchronous concurrency* $\Sigma_{conc}(\omega_{scc})$ (see Appendix A.5.4). As in *Quartz*, it is worth noting that they execute their *macro steps* synchronously in lockstep, i.e., statements can interact during concurrent execution.

Moreover, states in control-flow oriented SCCharts can be aborted and called in this context *abort transitions* $\Sigma_{abort}(\omega_{scc'})$, where this thesis considers *regular abortions* $\Sigma^{reg}_{abort}(\omega_{scc})$ and *immediate abortions* $\Sigma^{imm}_{abort}(\omega_{scc})$ (see Appendix A.5.2).

Control-flow oriented SCCharts are able to express different variants of conditions $\Sigma_{cond}(\omega_{scc})$ (see Appendix A.5.5) and loop variants $\Sigma_{loop}(\omega_{scc})$ (see Appendix A.5.7) by reconstructing the control flow using states and transitions [SLH16], where this thesis considers the condition variants introduced in Section 3.2.8 and loop variants introduced in Section 3.2.9.

Furthermore, there are final states in control-flow oriented models that stop the sequential execution, which are called *halt* statements $\Sigma_{halt}(\omega_{scc})$ in this thesis (see Appendix A.5.6).

This thesis focuses on data-flow oriented SCCharts that are able to invoke data-flow and control-flow oriented SCCharts. This is possible through the use of $\Sigma_{inv}(\omega_{scd})$ statements (see Appendix A.4.5).

Additionally, in control-flow oriented SCCharts, there are transitions that don't perform any action and consume no time when switching between states. These transitions are called *immediate transitions* $\Sigma_{nothing}(\omega_{scc})$ in this thesis (see Appendix A.5.8) to simplify readability and comparability with the constructs in the other models.

The consumption of time, as in SCL and *Quartz* models, can be explicitly programmed in control-flow oriented SCCharts by a transition between two states that consumes one unit of time. These constructs are called *pause transitions* $\Sigma_{pause}(\omega_{scc})$ in this thesis (see Appendix A.5.9).

Finally, a set of data-flow oriented SCChart statements is called a sequence $\Sigma_{seq}(\omega_{scd})$, where each statement ends with a semicolon as explained in the context of assignments (see Appendix A.3.10). In contrast, in control-flow oriented SCCharts, a set of states represents a sequence $\Sigma_{seq}(\omega_{scc})$ (see Appendix A.3.10). By enforcing a sequential execution of synchronous threads in data-flow oriented SCCharts and a sequential order of states in control-flow oriented SCCharts, both sequences are executed in a sequential manner.

The syntax and semantics defined (or referenced) in this chapter are used in the following chapters to present various approaches to transforming and optimizing the software models under consideration. In particular, they are often used to prove correctness.

Chapter 4

Model Transformation of ST Models to *Quartz* Models

Contents

As a first approach to reusing existing POUs in model-based design is the transformation of ST models into *Quartz* models, where the goal is to create a robust set of translation functions that ensure semantic preservation during the transition. In addition to the approaches presented in [WS21; WS22; WS24a; WS24b], it considers the following additional issues:

- **Model Declaration**: Mimicking the termination behavior of the initial model, i.e., distinguishing between models with and without memory
- **Interfaces and Variables**: Additional interfaces for event-driven execution control and an external controlled time (provided as an unbounded integer)

- **Data Types and Fields**: Additional IEC 61131-3 data types
- **POU invocations**: Instantiation and invocation of user-defined models, taking into account their individual termination behavior

The correctness of the translation functions is proved by theoretical reasoning, which includes a detailed analysis of the resulting syntax and semantics compared to the syntax rules and semantics specified in Chapter 3. In addition, the theoretical results are evaluated with real-world and self-defined ST models.

This chapter is structured as follows: Section 4.1 introduces the high-level design flow and translation strategy. Section 4.2 defines the translation functions and theoretical analysis. Section 4.3 presents an evaluation of the theoretical results, and Section 4.4 summarizes the transformation.

4.1. High-Level Design Flow – ST-to-Quartz

The high-level design flow for transforming an ST model $\omega_{st}^{\varphi} \in \Omega_{st}^{\varphi}$ of the variant *function block* ($\varphi = fb$), *function* ($\varphi = fun$), or *program* ($\varphi = prg$) to a *Quartz* model $\omega_{qrz} \in \Omega_{qrz}$ is shown in Figure 4.1.

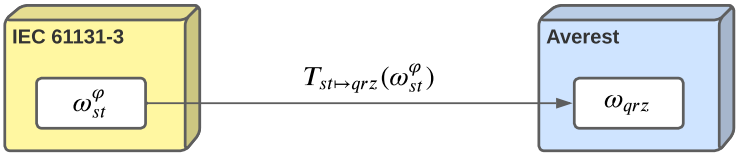


Figure 4.1.: *High-level design flow of the ST-to-Quartz transformation*

The ST-to-Quartz transformation $T_{st \mapsto qrz}(\omega_{st}^{\varphi})$ includes the following transformation steps:

1. $t_{st \mapsto qrz}^{\delta_\omega}(\delta_\omega(\omega_{st}^{\varphi}))$: Model declaration (see Section 4.2.1)
2. $t_{st \mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$: Interfaces (see Section 4.2.2)
3. $t_{st \mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$: Variables (see Section 4.2.3)
4. $t_{st \mapsto qrz}^{\alpha}(\alpha^{[+]}(\omega_{st}^{\varphi}))$: Data types and fields (see Section 4.2.4)
5. $t_{st \mapsto qrz}^{\Delta_{imports}}(\omega_{st}^{\varphi})$: POU imports (see Section 4.2.5)
6. $t_{st \mapsto qrz}^{\tau}(\tau(\omega_{st}^{\varphi}))$: Expressions (see Section 4.2.6)
7. $t_{st \mapsto qrz}^{\Sigma_{inv}}(\sigma_{inv}^{\vartheta}(\omega_{st}^{\varphi}))$: POU invocations (see Section 4.2.7)
8. $t_{st \mapsto qrz}^{\Sigma_{ass}}(\sigma_{ass}^{\vartheta}(\omega_{st}^{\varphi}))$: Assignments (see Section 4.2.8)
9. $t_{st \mapsto qrz}^{\Sigma_{cond}}(\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi}))$: Conditions (see Section 4.2.9)
10. $t_{st \mapsto qrz}^{\Sigma_{loop}}(\sigma_{loop}^{\vartheta}(\omega_{st}^{\varphi}))$: Loops (see Section 4.2.10)
11. $t_{st \mapsto qrz}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{qrz'}))$: Sequences (see Section 4.2.11)

Translation Strategy:

This chapter introduces two translation strategies illustrated in Figure 4.2: The ST model is either transformed into a *Quartz* model that (1) terminates after a finite number of *macro steps* $n \geq 0$ to mimic a POU without memory, or (2) runs in an infinite loop to mimic a POU with memory, where an iteration contains a finite number of *macro steps* $n \geq 1$[1]. Figure 4.2a shows the high-level runtime behavior of the resulting *Quartz* model that follows the first translation strategy, where the set of input variables `IN` is read when the *Quartz* model is invoked in *macro step* S_i. In the following *macro steps* during execution, the interfaces are synchronized, where only `CLK` may be updated externally and is read. Neither input variables `IN` are allowed to be updated nor output variables `OUT` are allowed to be processed externally until the *Quartz* model is terminated. In the final *macro step* S_m, `OUT` is returned to the invoking model for external processing. In contrast, Figure 4.2b shows the high-level runtime behavior of the resulting *Quartz* model that follows the second translation strategy, where the model is initialized in the first *macro step* S_0 and then waits until an iteration is triggered externally via the event input variable `EI`. When an iteration is triggered in *macro step* S_i, the set of input variables `IN` and an external clock variable `CLK` are read and can be processed. In the following *macro steps* during execution, the interfaces are synchronized, where only `CLK` may be updated externally. Neither input variables `IN` are allowed to be updated nor output variables `OUT` are allowed to be processed externally until the iteration reaches the *macro step* S_m. In this *macro step*, `OUT` and `EO` are returned to the invoking model for external processing. The subsequent final *macro step* is used to switch to the next iteration (triggered in the next PLC cycle).

Challenges:

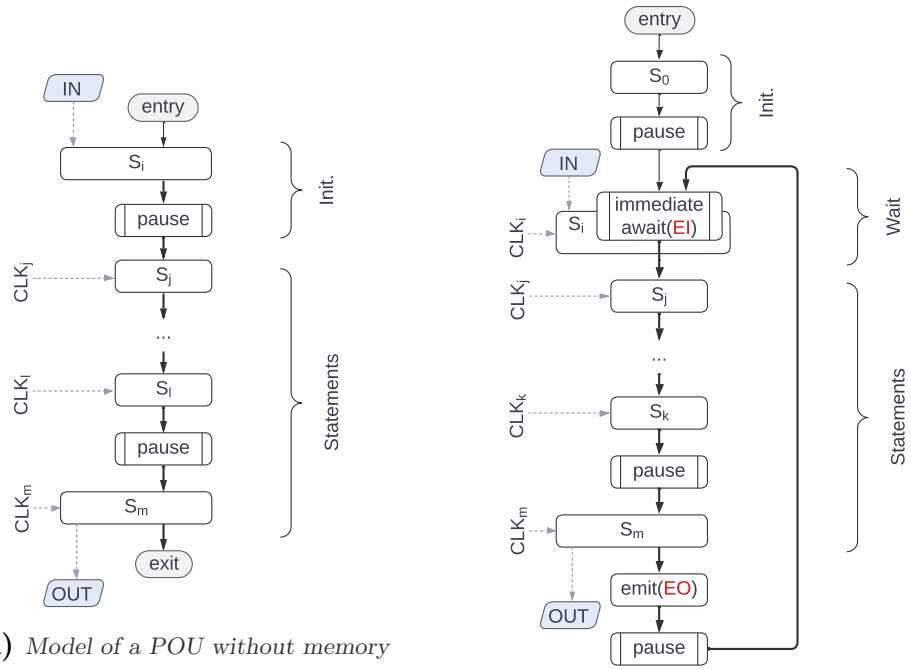
From this, the following challenges for translating ST models to *Quartz* models can be derived:

1. Cyclic execution of *Quartz* models (with and without memory)
2. Event-driven execution of synchronous parallel threads
3. Sequential execution of synchronous parallel threads
4. Dynamic system time
5. Translation of ST language constructs

4.2. From ST Models to *Quartz* Models

This section defines the individual translation functions for translating an ST model $\omega_{st}^{\varphi} \in \Omega_{st}^{\varphi}$ to a *Quartz* model $\omega_{qrz'} \in \Omega_{qrz}$ and analyzes the theoretical correctness.

[1] $n \geq 1$, because the `pause` statement between two iterations is considered in this context



(a) *Model of a POU without memory*

(b) *Model of a POU with memory*

Figure 4.2.: *ST-to-Quartz translation strategies: high-level runtime behavior of the resulting* Quartz *models*

4.2.1. Model Declaration

This step covers the translation function for translating an ST model declaration $\delta_\omega(\omega_{st}^{\varphi})$ to a *Quartz* model declaration $\delta_\omega(\omega_{qrz'})$. According to the introduced translation strategies, the resulting *Quartz* model of an ST model, variant $\varphi \in \{fb, prg\}$, is executed in an infinite loop, reflecting the memory behavior of the original ST model. For this, in each PLC cycle, the *Quartz* model waits until a loop iteration is triggered by an event-driven input, where the termination of the iteration is returned by an event-driven output. In contrast, the resulting *Quartz* model of an ST model, variant $\varphi = fun$, is invoked and executed sequentially without a surrounding loop, mimicking an ST-based POU without memory.

Definition 4.1 (Model Declaration – ST-to-Quartz). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements.* $\delta_\omega(\omega_{st}^{\varphi})$ *is translated to* $\delta_\omega(\omega_{qrz'})$ *using the translation function* $t_{st\mapsto qrz}^{\delta_\omega}(\delta_\omega(\omega_{st}^{\varphi}))$*, which is described by Algorithm 1.*

Correctness

To check the correctness of Definition 4.1, the following lemma is used.

Algorithm 1 Translate model declaration – ST-to-Quartz

Input: $\delta_\omega(\omega_{st}^{\varphi})$
Output: $\delta_\omega(\omega_{qrz'})$
Translation Function $t_{st\mapsto qrz}^{\delta_\omega}(\delta_\omega(\omega_{st}^{\varphi}))$**:**

switch φ **do**
  case fun **do**

$$\delta_\omega(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} t_{st\mapsto qrz}^{\Delta_{imports}}(\omega_{st}^{\varphi}) \\ \texttt{module } a_n(\omega_{st}^{\varphi})(t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi}))\{ \\ \quad t_{st\mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi}) \\ \quad t_{st\mapsto qrz}^{\Sigma}(\omega_{st}^{\varphi}) \\ \} \end{array} \right\}$$

  end
  case $fb \vee prg$ **do**

$$\delta_\omega(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} t_{st\mapsto qrz}^{\Delta_{imports}}(\omega_{st}^{\varphi}) \\ \texttt{module } a_n(\omega_{st}^{\varphi})(t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi}))\{ \\ \quad t_{st\mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi}) \\ \quad \texttt{loop}\{ \\ \quad\quad \texttt{immediate await(EI);} \\ \quad\quad t_{st\mapsto qrz}^{\Sigma}(\omega_{st}^{\varphi}) \\ \quad\quad \texttt{emit(EO); pause;} \\ \quad \} \\ \} \end{array} \right\}$$

  end
end

Lemma 4.1. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t_{st\mapsto qrz}^{\delta_\omega}(\delta_\omega(\omega_{st}^{\varphi}))$ translates $\delta_\omega(\omega_{st}^{\varphi})$ to $\delta_\omega(\omega_{qrz'})$ as specified in Definition 4.1. $\delta_\omega(\omega_{qrz'})$ conforms to the syntax rules of $\delta_\omega(\omega_{qrz})$ and preserves the semantics of $\delta_\omega(\omega_{st}^{\varphi})$ regarding its termination behavior.*

Proof The validity of Lemma 4.1 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\delta_\omega(\omega_{qrz'})$ with the syntax rules of $\delta_\omega(\omega_{qrz})$ as specified in Section 3.3. While the equivalence for $\varphi = fun$ is given by the specification, the correctness for $\varphi \in \{fb, prg\}$ follows from the syntactical correct sequence of $\sigma_{loop}^{inf}(\omega_{qrz'})$, $\sigma_{await}^{imm}(\omega_{qrz'})$, $\sigma_{ass}^{imm}(\omega_{qrz'})$, and $\sigma_{pause}(\omega_{qrz'})$. Second, there are two cases to distinguish in order to check semantic correctness:

- **Case 1 ($\varphi = fun$):** $\delta_\omega(\omega_{qrz'})$ does not have a surrounding sequence of an additional loop with event-driven constructs. Thus, $[\![\delta_\omega(\omega_{qrz})]\!]_\xi$ preserves $[\![\delta_\omega(\omega_{st}^{\varphi})]\!]_\xi$ regarding its termination behavior (see Section 3.2 and 3.3).

- **Case 2** ($\varphi \in \{fb, prg\}$)**:** Given the SOS transition rules of $\sigma^{inf}_{loop}(\omega_{qrz})$, $\sigma^{imm}_{await}(\omega_{qrz})$, and $\sigma_{pause}(\omega_{qrz})$, $\delta_{\omega}(\omega_{qrz})$ results in an infinite loop whose iteration is executed immediately when the event `EI` becomes true and triggers `EO` at the end of an iteration. Iterations are separated by $\sigma_{pause}(\omega_{qrz})$. Consequently, $[\![\delta_{\omega}(\omega_{qrz})]\!]_{\xi}$ preserves $[\![\delta_{\omega}(\omega^{\varphi}_{st})]\!]_{\xi}$ regarding its termination behavior (see Section 3.2 and 3.3).

Illustrative Example for Lemma 4.1

Usage examples are shown below using snippets of the resulting *Quartz* models[2] in combination with a visualization of the high-level runtime behavior. The resulting *Quartz* model of example `ST_ALARM` in Figure 4.3 represents a model without memory, and the resulting *Quartz* model of example `ST_ASS_DEL` in Figure 4.4 represents a model that preserves the state of the last *macro step* of the current PLC cycle for the next iteration (i.e., for the next PLC cycle).

```
1  module ST_ALARM(
2      bool ?xSENSOR_L, ...){
3
4      ST_ALARM = ... ;
5  }
```

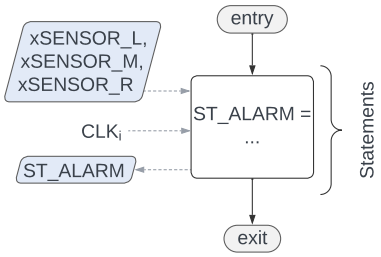


Figure 4.3.: *Resulting* Quartz *model of example* `ST_ALARM` *(without memory)*

```
1  module ST_ASS_DEL(
2    event bool ?EI,
3    event bool !EO){
4
5    int x0; ...
6    pause;
7
8    loop{
9      immediate await(EI);
10     next(y0) = y0 + x0;
11     pause;
12     emit(EO); pause;
13   }
14 }
```

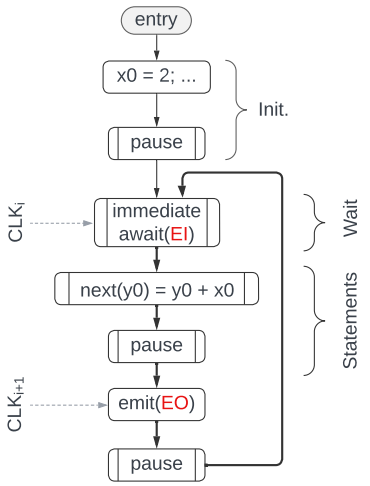


Figure 4.4.: *Resulting* Quartz *model of example* `ST_ASS_DEL` *(with memory)*

[2]Both *Quartz* models are included in Appendix C.2 and C.11.

4.2.2. Interfaces

This step covers the translation function for translating ST model interfaces $\Delta_{idcl}(\omega_{st}^{\varphi})$ to *Quartz* model interfaces $\Delta_{idcl}(\omega_{qrz'})$, where $\Delta_{idcl}(\omega_{st}^{\varphi}) = \Delta_{in}(\omega_{st}^{\varphi}) \cup \Delta_{out}(\omega_{st}^{\varphi}) \cup \Delta_{inout}(\omega_{st}^{\varphi})$.

Definition 4.2 (Interfaces – ST-to-Quartz). *Let $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. $\Delta_{idcl}(\omega_{qrz'})$ is derived from ω_{st}^{φ} and extended by event-driven variables and by an optional system time, using the translation function $t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$, which is described by Algorithm 2.*

Correctness

To check the correctness of Definition 4.2, the following lemmas are used.

Lemma 4.2. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, and ω_{qrz} always be invoked with connected $\Delta_{in}(\omega_{qrz})$. Then, for each interface $e_i \in \mathcal{E}_i(\omega_{st}^{\varphi})$, $t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ extends $\Delta_{in}(\omega_{qrz'})$, $\Delta_{out}(\omega_{qrz'})$, or $\Delta_{inout}(\omega_{qrz'})$, and adds possible assignments to defaults to $\Sigma_{seq}(\omega_{qrz'})$ as specified in Definition 4.2. $\Delta_{in}(\omega_{qrz'})$, $\Delta_{out}(\omega_{qrz'})$, and $\Delta_{inout}(\omega_{qrz'})$ conform to the syntax rules of $\Delta_{in}(\omega_{qrz})$, $\Delta_{out}(\omega_{qrz})$, and $\Delta_{inout}(\omega_{qrz})$ regarding the storage class, data type, and name. $\Sigma_{seq}(\omega_{qrz'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{qrz})$. With and without assignments to defaults, $\Delta_{in}(\omega_{qrz'})$, $\Delta_{out}(\omega_{qrz'})$, and $\Delta_{inout}(\omega_{qrz'})$ in combination with $\Sigma_{seq}(\omega_{qrz'})$ preserves the semantics of $\Delta_{in}(\omega_{st}^{\varphi})$, $\Delta_{out}(\omega_{st}^{\varphi})$, and $\Delta_{inout}(\omega_{st}^{\varphi})$ regarding information flow, modifiability, and initialization.*

Proof The validity of Lemma 4.2 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{in}(\omega_{qrz'})$, $\Delta_{out}(\omega_{qrz'})$, $\Delta_{inout}(\omega_{qrz'})$, and $\Sigma_{seq}(\omega_{qrz'})$ with the syntax rules of $\Delta_{in}(\omega_{qrz})$, $\Delta_{out}(\omega_{qrz})$, $\Delta_{inout}(\omega_{qrz})$, and $\Sigma_{seq}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of added interfaces $\mathcal{E}_i(\omega_{st}^{\varphi})$:

1. **Base Case:** When $\mathcal{E}_i(\omega_{st}^{\varphi}) = \varnothing$, there are no input variables, output variables with possible initialization, and inout variables, and thus no statements to add, which trivially conforms to the syntax rules of $\Delta_{in}(\omega_{qrz})$, $\Delta_{out}(\omega_{qrz})$, $\Delta_{inout}(\omega_{qrz})$, $\Sigma_{ass}^{imm}(\omega_{qrz})$, and $\Sigma_{pause}(\omega_{qrz})$, since these sets remain unchanged and are optional.
2. **Induction Hypothesis:** The lemma holds for any set of input variables, output variables with possible initialization, and inout variables.
3. **Inductive Step:** Adding an element to input variables, output variables with initialization, and inout variables results in an additional element in $\Delta_{in}(\omega_{qrz'})$, $\Delta_{out}(\omega_{qrz'})$, $\Delta_{inout}(\omega_{qrz'})$, $\Sigma_{ass}^{imm}(\omega_{qrz'})$, and

Algorithm 2 Translate interfaces – ST-to-Quartz

Input: ω_{st}^{φ}
Output: $\Delta_{in}(\omega_{qrz'})$, $\Delta_{out}(\omega_{qrz'})$, $\Delta_{inout}(\omega_{qrz'})$, $\Sigma_{seq}(\omega_{qrz'})$
Translation Function $t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$**:**

```
if φ ∈ {fb, prg} then
    Δ_in(ω_qrz') ← add event bool ?EI;
                                        ▷ add Boolean event input variable
    Δ_out(ω_qrz') ← add event bool !EO;
                                        ▷ add Boolean event output variable
end
if ω_st^φ contains time-based logic then
    Δ_in(ω_qrz') ← add nat ?CLK;
                                        ▷ add memorized input variable
end
if φ = fun ∧ e_rT ≠ ∅, where e_rT ∈ E_rT(ω_st^φ), E_rT(ω_st^φ) ⊂ E_i(ω_st^φ) then
    Δ_out(ω_qrz') ← add t^α_{st↦qrz}(α(e_rT)) !a_n(ω_st^φ);
                                        ▷ add memorized output variable
end
forall e_i ∈ E_i(ω_st^φ) do
    if e_i = e_iVs, where e_iVs ∈ E_iVs(ω_st^φ), E_iVs(ω_st^φ) ⊂ E_i(ω_st^φ) then
        Δ_in(ω_qrz') ← add t^α_{st↦qrz}(α(e_i)) ?a_n(e_i);
                                        ▷ add memorized input variable
    end
    if e_i = e_oVs, where e_oVs ∈ E_oVs(ω_st^φ), E_oVs(ω_st^φ) ⊂ E_i(ω_st^φ) then
        Δ_out(ω_qrz') ← add t^α_{st↦qrz}(α(e_i)) !a_n(e_i);
                                        ▷ add memorized output variable
        Σ_seq(ω_qrz') ← add a_n(e_i) = t^{τ_misc}_{st↦qrz}(π);
                                        ▷ add assignment to default value (if specified)
    end
    if e_i = e_iOVs, where e_iOVs ∈ E_iOVs(ω_st^φ), E_iOVs(ω_st^φ) ⊂ E_i(ω_st^φ) then
        Δ_inout(ω_qrz') ← add t^α_{st↦qrz}(α(e_i)) a_n(e_i);
                                        ▷ add memorized inout variable
    end
end
if Σ_seq(ω_qrz') ≠ ∅ then
    Σ_seq(ω_qrz') ← add pause;
                                        ▷ add pause if at least one variable is initialized
end
```

$\Sigma_{pause}(\omega_{qrz'})$. Their syntax still conforms to the syntax rules of $\Delta_{out}(\omega_{qrz})$, $\Delta_{inout}(\omega_{qrz})$, $\Sigma_{ass}^{imm}(\omega_{qrz})$, and $\Sigma_{pause}(\omega_{qrz})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{in}(\omega_{qrz})]\!]_{\xi}$, $[\![\Delta_{out}(\omega_{qrz})]\!]_{\xi}$, and $[\![\Delta_{inout}(\omega_{qrz})]\!]_{\xi}$ in combination with the SOS transition rules of $\Sigma_{ass}^{imm}(\omega_{qrz})$ and $\Sigma_{pause}(\omega_{qrz})$ (see Section 3.3) with $[\![\Delta_{in}(\omega_{st}^{\varphi})]\!]_{\xi}$, $[\![\Delta_{out}(\omega_{st}^{\varphi})]\!]_{\xi}$, and $[\![\Delta_{inout}(\omega_{st}^{\varphi})]\!]_{\xi}$ (see Section 3.2).

Illustrative Example for Lemma 4.2

As an example, below are some derived interfaces of the `ST_SIMPLE_PRG1` model[3]:

```
1    real ?PRG_C                    // added to Δ_in(ω_qrz')
2    real !PRG_OUT2                 // added to Δ_out(ω_qrz')
```

Lemma 4.3. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, prg\}$. Then, $t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ adds the additional event input `EI` to $\Delta_{in}(\omega_{qrz'})$ and the additional event output `EO` to $\Delta_{out}(\omega_{qrz'})$ as specified in Definition 4.2. $\Delta_{in}(\omega_{qrz'})$ and $\Delta_{out}(\omega_{qrz'})$ conform to the syntax rules of $\Delta_{in}(\omega_{qrz})$ and $\Delta_{out}(\omega_{qrz})$.*

Proof The validity of Lemma 4.3 is proved by comparing the resulting syntax of $\Delta_{in}(\omega_{qrz'})$ and $\Delta_{out}(\omega_{qrz'})$ with the syntax rules of $\Delta_{in}(\omega_{qrz})$ and $\Delta_{out}(\omega_{qrz})$ as specified in Section 3.3.

Illustrative Example for Lemma 4.3

As an example, below are the derived interfaces of the `ST_SIMPLE_PRG1` model[4]:

```
1    event bool ?EI                 // added to Δ_in(ω_qrz')
2    event bool !EO                 // added to Δ_out(ω_qrz')
```

Lemma 4.4. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, and ω_{st}^{φ} contains time-based logic, where time is a readable variable and is synchronized externally. Then, $t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ adds an additional memorized input to $\Delta_{in}(\omega_{qrz'})$ as specified in Definition 4.2. $\Delta_{in}(\omega_{qrz'})$ conforms to the syntax rules of $\Delta_{in}(\omega_{qrz})$.*

Proof The validity of Lemma 4.4 is proved by comparing the resulting syntax of $\Delta_{in}(\omega_{qrz'})$ with the syntax rules of $\Delta_{in}(\omega_{qrz})$ as specified in Section 3.3.

Illustrative Example for Lemma 4.4

As an example, below is the derived input of the `ST_SIMPLE_PRG1` model[5]:

```
1    nat ?CLK                       // added to Δ_in(ω_qrz')
```

[3] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.
[4] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.
[5] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.

Lemma 4.5. *Let ω_{st}^{fun} be translated to $\omega_{qrz'}$ and ω_{st}^{fun} has a specified return type, which is processed by ω_{st}^{fun}. Then, $t_{st\mapsto qrz}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ adds an additional memorized output for the specified return type to $\Delta_{out}(\omega_{qrz'})$ as specified in Definition 4.2. $\Delta_{out}(\omega_{qrz'})$ conforms to the syntax rules of $\Delta_{out}(\omega_{qrz})$.*

Proof The validity of Lemma 4.5 is proved by comparing the resulting syntax of $\Delta_{out}(\omega_{qrz'})$ with the syntax rules of $\Delta_{out}(\omega_{qrz})$ as specified in Section 3.3.

Illustrative Example for Lemma 4.5

As an example, below is the derived output of the `ST_SIMPLE_FUN` function in `ST_SIMPLE_PRG1` model[6]:

```
1    real !ST_SIMPLE_FUN              // added to Δout(ωqrz')
```

4.2.3. Variables

This step covers the translation function for translating local ST model variables $\Delta_{vdcl}(\omega_{st}^{\varphi})$ to local *Quartz* model variables $\Delta_{vdcl}(\omega_{qrz'})$, where $\Delta_{vdcl}(\omega_{st}^{\varphi}) = \Delta_{local}(\omega_{st}^{\varphi})$.

Definition 4.3 (Variables – ST-to-Quartz). *Let $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. $\Delta_{vdcl}(\omega_{qrz'})$ is derived from ω_{st}^{φ} and extended by event-driven interfaces of instances as well as additional instance output variables using the translation function $t_{st\mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$, which is described by Algorithm 3.*

Correctness

To check the correctness of Definition 4.3, the following lemmas are used.

Lemma 4.6. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, fun, prg\}$. Then, for each local variable $e_{lVs} \in \mathcal{E}_i(\omega_{st}^{\varphi})$ that is not derived from standard function blocks $e_d(e_{lVs}) = \varnothing$), $t_{st\mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$ adds a local variable to $\Delta_{local}(\omega_{qrz'})$ and a possible initialization to $\Sigma_{seq}(\omega_{qrz'})$ as specified in Definition 4.3. $\Delta_{local}(\omega_{qrz'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$ regarding the storage class, data type, and name, and $\Sigma_{seq}(\omega_{qrz'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{qrz})$. With and without initialization, $\Delta_{local}(\omega_{qrz'})$ in combination with $\Sigma_{seq}(\omega_{qrz'})$ preserves the semantics of $\Delta_{local}(\omega_{st}^{\varphi})$ regarding modifiability and initialization.*

[6] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.

Algorithm 3 Translate variables – ST-to-Quartz

Input: ω_{st}^{φ}
Output: $\Delta_{local}(\omega_{qrz'}), \Sigma_{seq}(\omega_{qrz'})$
Translation Function $t_{st\mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$**:**

forall $e_{b^{inst}} \in \mathcal{E}_{b^{inst}}(\omega_{st}^{\varphi})$ **do**
  $\Delta_{local}(\omega_{qrz'}) \leftarrow$ add `event bool` $a_n(a_{iN}(e_{b^{inst}}))$`_EI`;
    ▷ *add Boolean event input variable*
  $\Delta_{local}(\omega_{qrz'}) \leftarrow$ add `event bool` $a_n(a_{iN}(e_{b^{inst}}))$`_EO`;
    ▷ *add Boolean event output variable*
  forall $e_{oVs} \in \mathcal{E}_{oVs}(e_{b^{inst}})$ **do**
    $\Delta_{local}(\omega_{qrz'}) \leftarrow$ add $t_{st\mapsto qrz}^{\alpha}(\alpha(e_{oVs}))$ $a_n(a_{iN}(e_{b^{inst}}))\_a_n(e_{oVs})$;
      ▷ *add memorized variable for output variable*
  end
end
forall $e_{bfun'}$ *(derived from* ω_{st}^{φ}*)* **do**
  $\Delta_{local}(\omega_{qrz'}) \leftarrow$ add $t_{st\mapsto qrz}^{\alpha}(e_{rT}(e_{bfun'}))$ $a_n(e_{bfun'})\_\mathtt{1}i'(e_{bfun'})$;
    ▷ *add memorized variable for return type of* $e_{bfun'}$*, where* i' *represents the index of occurrence*
  forall $e_{oVs} \in \mathcal{E}_{oVs}(e_{bfun'})$ **do**
    $\Delta_{local}(\omega_{qrz'}) \leftarrow$ add $t_{st\mapsto qrz}^{\alpha}(\alpha(e_{oVs}))$ $a_n(e_{bfun'})\_a_n(e_{oVs})\_\mathtt{1}i'(e_{bfun'})$;
      ▷ *add memorized variable for output variable*
  end
end
forall $e_{lVs} \in \mathcal{E}_i(\omega_{st}^{\varphi})$ **do**
  if $e_d(e_{lVs}) = \varnothing$ **then**
    $\Delta_{local}(\omega_{qrz'}) \leftarrow$ add $t_{st\mapsto qrz}^{\alpha}(\alpha(e_{lVs}))$ $a_n(e_{lVs})$;
      ▷ *add memorized variable*
    $\Sigma_{seq}(\omega_{qrz'}) \leftarrow$ add $a_n(e_{lVs})$ `=` $t_{st\mapsto qrz}^{\tau_{misc}}(\pi)$;
      ▷ *add assignment to default value (if specified)*
  end
end
if $\Sigma_{seq}(\omega_{qrz'}) \neq \varnothing$ **then**
  $\Sigma_{seq}(\omega_{qrz'}) \leftarrow$ add `pause` (and remove `pause` added for initialized interfaces);
    ▷ *add* pause *if at least one variable is initialized*
end

Proof The validity of Lemma 4.6 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{local}(\omega_{qrz'})$ and $\Sigma_{seq}(\omega_{qrz'})$ with the syntax rules of $\Delta_{local}(\omega_{qrz})$ and $\Sigma_{seq}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of added variables $\mathcal{E}_i(\omega_{st}^{\varphi})$:

1. **Base Case:** When $\mathcal{E}_i(\omega_{st}^{\varphi}) = \varnothing$, there are no local variables and thus no statements to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$, $\Sigma_{ass}^{imm}(\omega_{qrz})$, and $\Sigma_{pause}(\omega_{qrz})$, since these sets remain un-

changed and are optional.

2. **Induction Hypothesis:** The lemma holds for any set of local variables with possible initialization.
3. **Inductive Step:** Adding an element to local variables with initialization results in an additional element in $\Delta_{local}(\omega_{qrz'})$, $\Sigma^{imm}_{ass}(\omega_{qrz'})$, and $\Sigma_{pause}(\omega_{qrz'})$. Their syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$, $\Sigma^{imm}_{ass}(\omega_{qrz})$, and $\Sigma_{pause}(\omega_{qrz})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{local}(\omega_{qrz})]\!]_\xi$ (in combination with the SOS transition rules of $\Sigma^{imm}_{ass}(\omega_{qrz})$ and $\Sigma_{pause}(\omega_{qrz})$) with $[\![\Delta_{local}(\omega^{\varphi}_{st})]\!]_\xi$ (see Section 3.2 and 3.3).

Illustrative Example for Lemma 4.6

As an example, below is the derived local variable of the `ST_SIMPLE_PRG1` model[7]:

```
1    int PRG_COUNT;                    // added to Δ_local(ω_arz')
2    PRG_COUNT = 4;                    // added to Σ_seq(ω_arz')
3    pause;                            // added to Σ_seq(ω_qrz')
```

Lemma 4.7. *Let ω^{φ}_{st} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, prg\}$. Then, for each instance $e_{b^{inst}} \in \mathcal{E}_{b^{inst}}(\omega^{\varphi}_{st})$, $t^{\Delta_{vdcl}}_{st \mapsto qrz}(\omega^{\varphi}_{st})$ adds the additional event variables for event-driven execution control to $\Delta_{local}(\omega_{qrz'})$ as specified in Definition 4.3. $\Delta_{local}(\omega_{qrz'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$ regarding the storage class, data type, and name that is consistently derived from the instance name.*

Proof The validity of Lemma 4.7 is checked by comparing the resulting syntax of $\Delta_{local}(\omega_{qrz'})$ with the syntax rules of $\Delta_{local}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of instances $\mathcal{E}_{b^{inst}}(\omega^{\varphi}_{st})$:

1. **Base Case:** When $\mathcal{E}_{b^{inst}}(\omega^{\varphi}_{st}) = \varnothing$, there are no local variables to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$, since this set remains unchanged and is optional.
2. **Induction Hypothesis:** The lemma holds for any set of instances $\mathcal{E}_{b^{inst}}(\omega^{\varphi}_{st})$.
3. **Inductive Step:** Adding an instance to $\mathcal{E}_{b^{inst}}(\omega^{\varphi}_{st})$ results in an additional element in $\Delta_{local}(\omega_{qrz'})$. Its syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$.

[7] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.

Illustrative Example for Lemma 4.7

As an example, below are the derived local variables of the `ST_SIMPLE_PRG1` model[8]:

```
1     event bool DEBOUNCE_01_EI;       // added to Δ_local(ω_qrz')
2     event bool DEBOUNCE_01_EO;       // added to Δ_local(ω_qrz')
```

Lemma 4.8. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, fun, prg\}$. Then, for each output $e_{oVs}(e_{b^{inst}}) \in \mathcal{E}_{oVs}(e_{b^{inst}})$ of each instance $e_{b^{inst}} \in \mathcal{E}_{b^{inst}}(\omega_{st}^{\varphi})$, $t_{st\mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$ adds a local variable to $\Delta_{local}(\omega_{qrz'})$ as specified in Definition 4.3. $\Delta_{local}(\omega_{qrz'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$ regarding the storage class, data type, and name that is consistently derived from the instance name.*

Proof The validity of Lemma 4.8 is checked by comparing the resulting syntax of $\Delta_{local}(\omega_{qrz'})$ with the syntax rules of $\Delta_{local}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of instances $\mathcal{E}_{b^{inst}}(\omega_{st}^{\varphi})$ with output variables $\mathcal{E}_{oVs}(e_{b^{inst}})$:

1. **Base Case:** When $\mathcal{E}_{b^{inst}}(\omega_{st}^{\varphi}) = \varnothing$ and $\mathcal{E}_{oVs}(e_{b^{inst}}) = \varnothing$, there are no local variables to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$, since this set remains unchanged and is optional.
2. **Induction Hypothesis:** The lemma holds for any set of instances with any set of output variables.
3. **Inductive Step:** Adding an instance to $\mathcal{E}_{b^{inst}}(\omega_{st}^{\varphi})$ with an output variable to $\mathcal{E}_{oVs}(e_{b^{inst}})$ results in an additional element in $\Delta_{local}(\omega_{qrz'})$. Its syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$.

Illustrative Example for Lemma 4.8

As an example, below are the derived local variables of the `ST_SIMPLE_PRG1` model[9]:

```
1     bool DEBOUNCE_01_OUT;        // added to Δ_local(ω_qrz')
2     nat DEBOUNCE_01_ET_OFF;      // added to Δ_local(ω_qrz')
```

Lemma 4.9. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, fun, prg\}$. Then, for the return type $e_{rT}(e_{b^{fun'}})$ and each output $e_{oVs}(e_{b^{fun'}}) \in \mathcal{E}_{oVs}(e_{b^{fun'}})$ of each user-defined function $e_{b^{fun'}} \in \mathcal{E}_{b^{fun'}}(\omega_{st}^{\varphi})$, $t_{st\mapsto qrz}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$ adds an additional variable to $\Delta_{local}(\omega_{qrz'})$ as specified in Definition 4.3. $\Delta_{local}(\omega_{qrz'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$ regarding the storage class,*

[8] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.
[9] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.

data type, and name that is consistently derived from the instance name and index of occurrence.

Proof The validity of Lemma 4.9 is checked by comparing the resulting syntax of $\Delta_{local}(\omega_{qrz'})$ with the syntax rules of $\Delta_{local}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of user-defined functions $\mathcal{E}_{bfun'}(\omega_{st}^{\varphi})$ with return type and output variables $\mathcal{E}_{oVs}(e_{bfun'})$:

1. **Base Case:** When $\mathcal{E}_{bfun'}(\omega_{st}^{\varphi}) = \varnothing$ and $\mathcal{E}_{oVs}(e_{bfun'}) = \varnothing$, there are no local variables to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$, since this set remains unchanged and is optional.
2. **Induction Hypothesis:** The lemma holds for any set of user-defined functions with any set of output variables.
3. **Inductive Step:** Adding a user-defined function with return type to $\mathcal{E}_{bfun'}(\omega_{st}^{\varphi})$ and an output variable to $\mathcal{E}_{oVs}(e_{bfun'})$ results in two additional elements in $\Delta_{local}(\omega_{qrz'})$ (one for the return type and one for the output). Its syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$.

Illustrative Example for Lemma 4.9

As an example, below are the derived local variables of the `ST_ASS_IMM3` model[10]:

```
1    int{32768} ST_ASS_IMM_OUT_y_11;      // added to Δ_local(ω_qrz')
2    bool ST_ASS_IMM_OUT_11;              // added to Δ_local(ω_qrz')
```

4.2.4. Data Types and Fields

This step covers the translation function for translating ST data types and fields $\mathcal{A}^{[+]}(\omega_{st}^{\varphi})$ to *Quartz* data types and fields $\mathcal{A}^{[+]}(\omega_{qrz'})$.

Definition 4.4 (Data types and fields – ST-to-Quartz). *Let $\alpha(\omega_{st}^{\varphi})$ be a considered ST data type and $\alpha^{+}(\omega_{st}^{\varphi})$ be an ST data type field. ST data types and fields $\mathcal{A}^{[+]}(\omega_{st}^{\varphi})$ are translated to* Quartz *data types and fields $\mathcal{A}^{[+]}(\omega_{qrz'})$ using the translation function $t^{\alpha}_{st\mapsto qrz}(\alpha^{[+]}(\omega_{st}^{\varphi}))$, which is described by Algorithm 37 in Appendix I.0.1*[11]*.*

Correctness

To check the correctness of Definition 4.4, the following lemma is used.

[10] Both models, ST and *Quartz*, are included in Appendix B.18 and C.18.

[11] This algorithm describes an intuitive mapping, which is moved to the appendix for the sake of readability, but is not necessary for understanding the following lemma.

Lemma 4.10. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, and* bit vector, integer, floating point, *and* duration *be the considered data type categories $\mathcal{A}^{[+]}(\omega_{st}^{\varphi})$ as specified in Section 3.2. Then, $t^{\alpha}_{st \mapsto qrz}(\alpha^{[+]}(\omega_{st}^{\varphi}))$ translates $\alpha^{[+]}(\omega_{st}^{\varphi})$ to $\alpha^{[+]}(\omega_{qrz'})$ as specified in Definition 4.4. $\alpha^{[+]}(\omega_{qrz'})$ conforms to the syntax rules of $\alpha^{[+]}(\omega_{qrz})$ and preserves the semantics of $\alpha^{[+]}(\omega_{st}^{\varphi})$ regarding boundaries, precision, resolution, and defaults (if applicable).*

Proof The validity of Lemma 4.10 is checked as follows: First, the syntactic correctness is proved by comparing all resulting data types and fields $\alpha^{[+]}(\omega_{qrz'})$ with syntax rules $\alpha^{[+]}(\omega_{qrz})$ as specified in Section 3.3. Second, the semantic correctness is checked by comparing $[\![\alpha^{[+]}(\omega_{qrz})]\!]_{\xi}$ with $[\![\alpha^{[+]}(\omega_{st}^{\varphi})]\!]_{\xi}$ (see Section 3.2 and 3.3), taking into account that the resolution of the duration data type is restricted to milliseconds.

Illustrative Example for Lemma 4.10

Usage examples are given by illustrative examples of previous lemmas, such as Lemma 4.9.

4.2.5. POU Imports

This step covers the translation function for translating instantiated and invoked POU imports in ST models to *Quartz* model imports $\Delta_{imports}(\omega_{qrz'})$.

Definition 4.5 (POU imports – ST-to-Quartz). *Let $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. Instantiated and invoked POUs in ST models are derived from external blocks $e_{lVs} \in \mathcal{E}(\omega_{st}^{\varphi}), e_d(e_{lVs}) \neq \varnothing$ and user-defined blocks $e_{bfun'} \in F'(\omega_{st}^{\varphi})$. These blocks are translated to* Quartz *model imports $\Delta_{imports}(\omega_{qrz'})$ using the translation function $t^{\Delta_{imports}}_{st \mapsto qrz}(\omega_{st}^{\varphi})$, which is described by Algorithm 4.*

Algorithm 4 Translate POU imports – ST-to-Quartz

Input: ω_{st}^{φ}
Output: $\Delta_{imports}(\omega_{qrz'})$
Translation Function $t^{\Delta_{imports}}_{st \mapsto qrz}(\omega_{st}^{\varphi})$**:**
  forall $e_{lVs} \in \mathcal{E}(\omega_{st}^{\varphi}), e_d(e_{lVs}) \neq \varnothing$ **do**
    $\Delta_{imports}(\omega_{qrz'}) \leftarrow$ add `import` $(e_d(e_{lVs}))$`.*`;
  end
  forall $e_{bfun'} \in F'(\omega_{st}^{\varphi})$ *(derived from ω_{st}^{φ})* **do**
    $\Delta_{imports}(\omega_{qrz'}) \leftarrow$ add `import` $a_n(e_{bfun'}))$`.*`;
  end

Correctness

To check the correctness of Definition 4.5, the following lemma is used.

Lemma 4.11. *Let the IEC 61131-3 standard function blocks (`RS`, `SR`, `TOF`, `TON`) [GDV14] be available as semantically and syntactically correct* Quartz *models. Then, for each instance* $e_{lVs} \in \mathcal{E}(\omega_{st}^{\varphi}), e_d(e_{lVs}) \neq \varnothing$ *and user-defined function* $e_{bfun'} \in F'(\omega_{st}^{\varphi})$, $t_{st\mapsto qrz}^{\Delta_{imports}}(\omega_{st}^{\varphi})$ *adds the corresponding import to* $\Delta_{imports}(\omega_{qrz'})$ *(if not already imported) as specified in Definition 4.5.* $\Delta_{imports}(\omega_{qrz'})$ *conforms to the syntax rules of* $\Delta_{imports}(\omega_{qrz})$.

Proof The validity of Lemma 4.11 is checked by comparing the resulting syntax of $\Delta_{imports}(\omega_{qrz'})$ with the syntax rules of $\Delta_{imports}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of instances $\mathcal{E}(\omega_{st}^{\varphi}) = \{e_{lVs} \mid e_d(e_{lVs}) \neq \varnothing\}$ and user-defined functions $F'(\omega_{st}^{\varphi})$:

1. **Base Case:** When $\mathcal{E}(\omega_{st}^{\varphi}) = \varnothing$ and $F'(\omega_{st}^{\varphi}) = \varnothing$, there are no modules to import, which trivially conforms to the syntax rules of $\Delta_{imports}(\omega_{qrz})$, since this set is optional.
2. **Induction Hypothesis:** The lemma holds for any set of instances and any set of user-defined functions.
3. **Inductive Step:** Adding an instance and user-defined function to $\mathcal{E}(\omega_{st}^{\varphi})$ and $F'(\omega_{st}^{\varphi})$ results in two additional elements in $\Delta_{imports}(\omega_{qrz'})$ (one for the instance and one for the user-defined function). Its syntax still conforms to the syntax rules of $\Delta_{imports}(\omega_{qrz})$.

Illustrative Example for Lemma 4.11

As an example, below are the derived imports of the `ST_SIMPLE_PRG1` model[12]:

```
1    import ST_DEBOUNCE.*;        // added to Δ_imports(ω_qrz')
2    import ST_SIMPLE_FUN.*       // added to Δ_imports(ω_qrz')
```

4.2.6. Expressions

This step covers the translation function for translating expressions in ST models $\mathcal{T}(\omega_{st}^{\varphi})$ to expressions in *Quartz* models $\mathcal{T}(\omega_{qrz'})$.

Definition 4.6 (Expressions – ST-to-Quartz). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements. An expression in ST models* $\tau(\omega_{st}^{\varphi}) \in \mathcal{T}(\omega_{st}^{\varphi})$ *is translated to an expression in* Quartz *models* $\tau(\omega_{qrz'}) \in \mathcal{T}(\omega_{qrz'})$ *using the translation function* $t_{st\mapsto qrz}^{\tau}(\tau(\omega_{st}^{\varphi}))$*, which is described by Algorithm 38 in Appendix I.0.2*[13].

[12] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.

[13] This algorithm describes an intuitive mapping, which is moved to the appendix for the sake of readability, but is not necessary for understanding the following lemma.

Correctness

To check the correctness of Definition 4.6, the following lemma is used.

Lemma 4.12. *Let* ω_{st}^{φ} *be translated to* $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, *and* miscellaneous, compare operators, arithmetic operators, conditional operator, *and* boolean operators *be the considered expression categories* $\mathcal{T}(\omega_{st}^{\varphi})$ *as specified in Section 3.2. Then, for each* $\tau(\omega_{st}^{\varphi}) \in \mathcal{T}(\omega_{st}^{\varphi})$, $t^{\tau}_{st \mapsto qrz}(\tau(\omega_{st}^{\varphi}))$ *translates* $\tau(\omega_{st}^{\varphi})$ *to* $\tau(\omega_{qrz'})$ *as specified in Definition 4.6.* $\tau(\omega_{qrz'})$ *conforms to the syntax rules of* $\tau(\omega_{qrz})$ *and preserves the semantics of* $\tau(\omega_{st}^{\varphi})$ *regarding the type system and SOS transition rules.*

Proof The validity of Lemma 4.12 is proved as follows: First, the syntactic correctness is checked by comparing all resulting expressions $\tau(\omega_{qrz'})$ with syntax rules $\tau(\omega_{qrz})$ as specified in Section 3.3. Second, the semantic correctness is checked by comparing $[\![\tau(\omega_{qrz})]\!]_{\xi}$ with $[\![\tau(\omega_{st}^{\varphi})]\!]_{\xi}$, their type system, and SOS transition rules as specified in Section 3.2 and 3.3. As a result, each considered ST expression can be mapped to a corresponding *Quartz* expression.

Illustrative Example for Lemma 4.12

As an example, below are two derived expressions of the `ST_SIMPLE_FUN` model[14]:

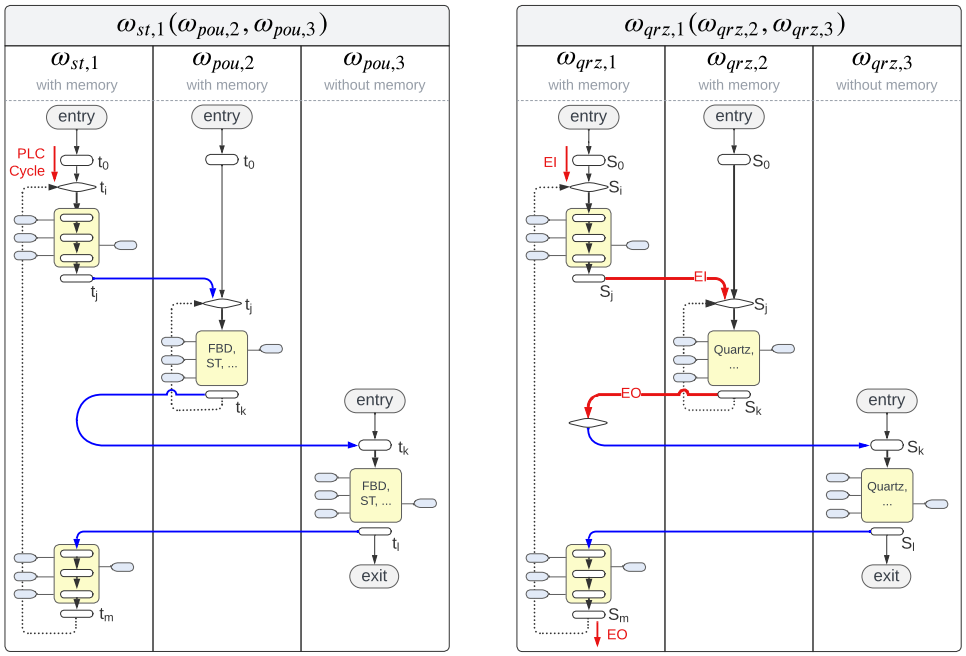
```
1    (COUNT+1)          // result of t^τ_{st↦arz}(COUNT+1)
2    ((A1*B1)/C1)       // result of t^τ_{st↦qrz}((A1*B1)/C1)
```

4.2.7. POU Invocations

This step covers the translation function for translating POU invocations in ST models $\Sigma_{inv}^{\vartheta}(\omega_{st}^{\varphi})$ to *Quartz* model invocations in *Quartz* models $\Sigma_{inv}^{\vartheta}(\omega_{qrz'})$. The high-level runtime behavior of an ST model with memory translated to a *Quartz* model mimicking memory behavior is illustrated in Figure 4.5. Figure 4.5a shows the runtime behavior of an example ST model that is triggered in each PLC cycle (red) and invokes a POU with memory and a POU without memory depending on their execution order (blue). In contrast, Figure 4.5b shows the runtime behavior of the resulting *Quartz* model, whose model with memory is triggered by an additional event-driven variable (red) and the model without memory without an additional event-driven variable, depending on its execution order (blue).

Definition 4.7 (POU Invocations – ST-to-Quartz). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements and POU invocations in ST models* $\Sigma_{inv}(\omega_{st}^{\varphi})$ *are given as* complete formal function call *[GDV14]. A*

[14] Both models, ST and *Quartz*, are included in Appendix B.26 and C.26.



(a) *Initial ST model* **(b)** *Resulting* Quartz *model*

Figure 4.5.: *High-level runtime behavior of a model with memory that invokes two models (Approach: ST-to-Quartz)*

POU invocation $\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}) \in \Sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})$ *is translated to a* Quartz *model invocation* $\sigma^{\vartheta}_{inv}(\omega_{qrz'}) \in \Sigma^{\vartheta}_{inv}(\omega_{qrz'})$ *using the translation function* $t^{\Sigma_{inv}}_{st \mapsto qrz}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))$*, which is described by Algorithm 5.*

Correctness

To check the correctness of Definition 4.7, the following lemmas are used.

Lemma 4.13. *Let* ω^{φ}_{st} *be translated to* $\omega_{qrz'}$ *and* $\varphi \in \{fb, prg\}$*. Then, for an invoked instance* $\sigma^{fb}_{inv}(\omega^{\varphi}_{st})$*,* $t^{\Sigma_{inv}}_{st \mapsto qrz}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st}))$ *adds an event-driven control sequence to* $\Sigma_{seq}(\omega_{qrz'})$ *as specified in Definition 4.7. Furthermore,* $t^{\Sigma_{inv}}_{st \mapsto qrz}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st}))$ *adds a synchronous concurrent thread to invoke instance with related arguments including event-driven interfaces and synchronized clock (if specified) as specified in Definition 4.7.* $\Sigma_{seq}(\omega_{qrz'})$ *conforms to the syntax rules of* $\Sigma_{seq}(\omega_{qrz})$ *and* $\sigma^{fb}_{inv}(\omega_{qrz'})$ *to the syntax rules of* $\sigma^{fb}_{inv}(\omega_{qrz})$*. The combination of* $\Sigma_{seq}(\omega_{qrz})$ *and* $\sigma^{fb}_{inv}(\omega_{qrz})$ *preserves the SOS transition rules of* $\sigma^{fb}_{inv}(\omega^{\varphi}_{st})$ *regarding termination behavior.*

Proof The validity of Lemma 4.13 is proved by comparing the resulting syntax of $\Delta_{seq}(\omega_{qrz'})$ with the syntax rules of $\Delta_{seq}(\omega_{qrz})$, and $\sigma^{fb}_{inv}(\omega_{qrz'})$ with

Algorithm 5 Invoke POU – ST-to-Quartz

Input: $\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})$
Output: $\Sigma_{seq}(\omega_{qrz'})$, $\sigma^{fb}_{inv}(\omega_{qrz'})$, $\sigma^{f'}_{inv}(\omega_{qrz'})$
Translation Function $t^{\Sigma_{inv}}_{st\mapsto qrz}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))$**:**

switch ϑ **do**

case fb **do**

$\Sigma_{seq}(\omega_{qrz'}) \leftarrow$ add $\left\{ \begin{array}{l} \texttt{emit}((a_{iN}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st})))\texttt{\_EI}); \\ \texttt{immediate await}((a_{iN}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st})))\texttt{\_EO}); \end{array} \right\}$

▷ *add event-driven control sequence*

$\sigma^{fb}_{inv}(\omega_{qrz'}) \leftarrow$ `||` $(a_{iN}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st})))$`:`
$(a_{tN}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st})))$`(`$(a_{iN}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st})))$`_EI, `$(a_{iN}(\sigma^{fb}_{inv}(\omega^{\varphi}_{st})))$`_EO,`
`[CLK,]` $\mathcal{I}$`, `$\mathcal{O}$`);`

▷ *add thread with event-driven interfaces and system time (if specified)*

end

case f' **do**

$\sigma^{f'}_{inv}(\omega_{qrz'}) \leftarrow (a_{tN}(\sigma^{f'}_{inv}(\omega^{\varphi}_{st})))$`([CLK,]` $\mathcal{I}$`, `$\mathcal{O}$`);`

▷ *add invocation with clock (if specified)*

end

forall $i \in \mathcal{I}, \mathcal{I} = \mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) \cup \mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ **do**

$i \mapsto t^{\tau}_{st\mapsto qrz}(i)$;

▷ *add translated input or inout argument*

end

forall $o \in \mathcal{O}, \mathcal{O} = \mathcal{E}_{oVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ **do**

$o \leftarrow o(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))))$;

▷ *add translated output argument*

end

end

the syntax rules of $\sigma^{fb}_{inv}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of input variables including inout variables $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) \cup \mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ and output variables $\mathcal{E}_{oVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ of an invoked instance $\sigma^{fb}_{inv}(\omega^{\varphi}_{st})$ with system time:

1. **Base Case:** When $\sigma^{fb}_{inv}(\omega^{\varphi}_{st}) \neq \varnothing$, $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) = \varnothing$, $\mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) = \varnothing$ and $\mathcal{E}_{oVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) = \varnothing$, there are no existing interfaces to add, but added sequence $\sigma^{imm}_{ass}(\omega_{qrz'})$, $\sigma^{imm}_{await}(\omega_{qrz'})$, and $\sigma^{fb}_{inv}(\omega_{qrz'})$, which conforms to the syntax rules of $\sigma^{imm}_{ass}(\omega_{qrz})$, $\sigma^{imm}_{await}(\omega_{qrz})$, and $\sigma^{fb}_{inv}(\omega_{qrz})$.
2. **Induction Hypothesis:** The lemma holds for any set of input variables including inout variables and any set of output variables of an invoked instance.
3. **Inductive Step:** Adding an input, inout variable, and output variable to $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$, $\mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$, and

$\mathcal{E}_{oVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ results in three additional interfaces and sequence $\sigma^{imm}_{ass}(\omega_{qrz'})$, $\sigma^{imm}_{await}(\omega_{qrz'})$, and $\sigma^{fb}_{inv}(\omega_{qrz'})$, which still conforms to the syntax rules of $\sigma^{imm}_{ass}(\omega_{qrz})$, $\sigma^{imm}_{await}(\omega_{qrz})$, and $\sigma^{fb}_{inv}(\omega_{qrz})$.

Second, the semantic correctness is checked by comparing the SOS transition rules of the sequence $\sigma^{imm}_{ass}(\omega_{qrz})$, $\sigma^{imm}_{await}(\omega_{qrz})$, and $\sigma^{fb}_{inv}(\omega_{qrz})$ with the SOS transition rules of $\sigma^{fb}_{inv}(\omega^{\varphi}_{st})$. Triggering `EI` and waiting for `EO` mimics a block invocation in ST models.

Illustrative Example for Lemma 4.13

As an example, below is a derived event-driven sequence and model invocation including arguments in the `ST_SIMPLE_PRG1` model[15]:

```
1    loop{ ...
2        emit(DEBOUNCE_01_EI);              // event input
3        immediate await(DEBOUNCE_01_EO); // event output
4        ... }
5    || DEBOUNCE_01:DEBOUNCE(               // add. thread
6        DEBOUNCE_01_EI,                    // event input
7        DEBOUNCE_01_EO,                    // event output
8        CLK,                               // clock
9        PRG_IN, 2000,                      // in(out) arg.
10       DEBOUNCE_01_OUT,                   // output arg.
11       DEBOUNCE_01_ET_OFF);               // output arg.
```

Lemma 4.14. *Let ω^{φ}_{st} be translated to $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, and user-defined function calls are designed as complete formal function call. Then, $t^{\Sigma_{inv}}_{st\mapsto qrz}(\sigma^{f'}_{inv}(\omega^{\varphi}_{st}))$ translates a user-defined function call $\sigma^{f'}_{inv}(\omega^{\varphi}_{st})$ to a* Quartz *model invocation $\sigma^{f'}_{inv}(\omega_{qrz'})$ with related arguments including synchronized clock (if specified) as specified in Definition 4.7. $\sigma^{f'}_{inv}(\omega_{qrz'})$ conforms to the syntax rules of $\sigma^{f'}_{inv}(\omega_{qrz})$ and preserves the SOS transition rules of $\sigma^{f'}_{inv}(\omega^{\varphi}_{st})$ regarding termination behavior.*

Proof The validity of Lemma 4.14 is checked by comparing the resulting syntax of $\sigma^{f'}_{inv}(\omega_{qrz'})$ with the syntax rules of $\sigma^{f'}_{inv}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of input variables including inout variables $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) \cup \mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ and output variables $\mathcal{E}_{oVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ of an invoked user-defined function $\sigma^{f'}_{inv}(\omega^{\varphi}_{st})$ with system time:

1. **Base Case:** When $\sigma^{f'}_{inv}(\omega^{\varphi}_{st}) \neq \varnothing$, $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) = \varnothing$, $\mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) = \varnothing$ and $\mathcal{E}_{oVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st}))) = \varnothing$, there are no existing interfaces to add, which conforms to the syntax rules of $\sigma^{f'}_{inv}(\omega_{qrz})$.

[15] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.

2. **Induction Hypothesis:** The lemma holds for any set of input variables including inout variables and any set of output variables of an invoked user-defined function.
3. **Inductive Step:** Adding an input, inout variable, and output variable to $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$, $\mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$, and $\mathcal{E}_{oVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{st})))$ results in three additional interfaces, which still conforms to the syntax rules $\sigma^{f'}_{inv}(\omega_{qrz})$.

Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma^{f'}_{inv}(\omega_{qrz})$ with the SOS transition rules of $\sigma^{f}_{inv}(\omega^{\varphi}_{st})$. An inline invocation mimics a block invocation in ST models, which can contain multiple *macro steps*.

Illustrative Example for Lemma 4.14

As an example, below is a derived model invocation including arguments in the `ST_SIMPLE_PRG1` model[16]:

```
1    loop{ ...
2        ST_SIMPLE_FUN(                          // invocation
3            (PRG_A + 2.0),                      // in(out) arg.
4            PRG_B, PRG_C, PRG_COUNT,            // in(out) arg.
5            ST_SIMPLE_FUN_l1);                  // output arg.
6        ... }
```

4.2.8. Assignments

This step covers the translation function for translating assignments in ST models $\Sigma^{\vartheta}_{ass}(\omega^{\varphi}_{st})$ to assignments in *Quartz* models $\Sigma^{\vartheta}_{ass}(\omega_{qrz'})$.

Definition 4.8 (Translation of assignments – ST-to-Quartz). *Let $\Omega^{\varphi}_{st} = \{\omega^{\varphi}_{st} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements and $\Sigma^{\vartheta}_{ass}(\omega^{\varphi}_{st})$ be the set of assigned variables. An immediate assignment $\sigma^{imm}_{ass}(\omega^{\varphi}_{st}) \in \Sigma^{imm}_{ass}(\omega^{\varphi}_{st})$ (i.e., $rhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))$ does not depend on $lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))$ and a delayed assignment $\sigma^{del}_{ass}(\omega^{\varphi}_{st}) \in \Sigma^{del}_{ass}(\omega^{\varphi}_{st})$ (i.e., $rhs(\sigma^{del}_{ass}(\omega^{\varphi}_{st}))$ does depend on $lhs(\sigma^{del}_{ass}(\omega^{\varphi}_{st}))$ in ST models is translated to a sequence $\Sigma_{seq}(\omega_{qrz'})$ in* Quartz *models using the translation function $t^{\Sigma_{ass}}_{st \mapsto qrz}(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{st}))$, which is described by Algorithm 6.*

Correctness

To check the correctness of Definition 4.8, the following lemmas are used.

Lemma 4.15. *Let ω^{φ}_{st} be translated to $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, and $\vartheta = imm$. Then, $t^{\Sigma_{ass}}_{st \mapsto qrz}(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))$ translates $\sigma^{imm}_{ass}(\omega^{\varphi}_{st})$ to a sequence $\Sigma_{seq}(\omega_{qrz'})$ of $\sigma_{pause}(\omega_{qrz'})$ and $\sigma^{imm}_{ass}(\omega_{qrz'})$ as specified in Definition 4.8,*

[16] Both models, ST and *Quartz*, are included in Appendix B.27 and C.27.

Algorithm 6 Translate assignment – ST-to-Quartz

Input: $\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{st})$
Output: $\Sigma_{seq}(\omega_{qrz'})$
Translation Function $t^{\Sigma_{ass}}_{st\mapsto qrz}(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{st}))$**:**

switch ϑ **do**
  case *imm* **do**
    $\Sigma_{seq}(\omega_{qrz'}) \leftarrow \text{add} \left\{ \begin{array}{l} \texttt{pause;} (\text{if } lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st})) \text{ was updated} \\ \quad \text{or read earlier in the current } \textit{macro step}) \\ lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st})) \texttt{ = } t^{\tau}_{st\mapsto qrz}(rhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))); \end{array} \right\}$
  end
  case *del* **do**
    $\Sigma_{seq}(\omega_{qrz'}) \leftarrow \text{add} \left\{ \begin{array}{l} \texttt{next(}lhs(\sigma^{del}_{ass}(\omega^{\varphi}_{st}))\texttt{) =} \\ \quad t^{\tau}_{st\mapsto qrz}(rhs(\sigma^{del}_{ass}(\omega^{\varphi}_{st}))); \\ \texttt{pause;} \end{array} \right\}$
  end
end

considering the last update and read of $lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))$*.* $\Sigma_{seq}(\omega_{qrz'})$ *conforms to the syntax rules of* $\Sigma_{seq}(\omega_{qrz})$*, preserves the SOS transition rules of* $\sigma^{imm}_{ass}(\omega^{\varphi}_{st})$*, and respects the single assignment per* macro step *constraint in* Quartz*.*

Proof The validity of Lemma 4.15 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{qrz'})$ with the syntax rules of $\Sigma_{seq}(\omega_{qrz})$ as specified in Section 3.3, where there are two cases to distinguish:

- **Case 1 (**$lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))$ **was updated or read earlier in the current *macro step*):** $\Sigma_{seq}(\omega_{qrz'}) = \{\sigma_{pause}(\omega_{qrz'}); \sigma^{imm}_{ass}(\omega_{qrz'})\}$, which conforms to the syntax rules of $\sigma_{pause}(\omega_{qrz})$ and $\sigma^{imm}_{ass}(\omega_{qrz})$.
- **Case 2 (**$lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))$ **was neither updated nor read earlier in the current *macro step*):** $\Sigma_{seq}(\omega_{qrz'}) = \{\sigma^{imm}_{ass}(\omega_{qrz'})\}$, which conforms to the syntax rules of $\sigma^{imm}_{ass}(\omega_{qrz})$.

Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{pause}(\omega_{qrz})$ in combination with $\sigma^{imm}_{ass}(\omega_{qrz})$ (or $\sigma^{imm}_{ass}(\omega_{qrz})$ only, respectively) with the SOS transition rules of $\sigma^{imm}_{ass}(\omega^{\varphi}_{st})$. The semantics are preserved, considering that a PLC cycle can contain several *macro steps* (see Section 3.2 and 3.3). The single assignment per *macro step* is respected by considering the last update and read of $lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{st}))$.

Illustrative Example for Lemma 4.15

As an example, the immediately assigned variables of the `ST_ASS_IMM1` model and `ST_ASS_IMM2` model are translated as follows[17]:

```
1   y:=x;
2       ⇒ y=x;                  // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
3   ------------------------
4   y0:=x0; y1:=x1;
5       ⇒ y0=x0;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
6       ⇒ y1=x1;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
7   ------------------------
8   y0:=x0; y1:=x1; y0:=x2;
9       ⇒ y0=x0;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
10      ⇒ y1=x1;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
11      ⇒ pause;                // σ_pause(ω_qrz') added to Σ_seq(ω_qrz')
12      ⇒ y0=x2;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
13  ------------------------
14  y2:=x0; y2:=x1;
15      ⇒ y2=x0;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
16      ⇒ pause;                // σ_pause(ω_qrz') added to Σ_seq(ω_qrz')
17      ⇒ y2=x1;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
18  ------------------------
19  y0:=x0; x0:=y0+x1;
20      ⇒ y0=x0;                // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
21      ⇒ pause;                // σ_pause(ω_qrz') added to Σ_seq(ω_qrz')
22      ⇒ x0=y0+x1;             // σ^imm_ass(ω_qrz') added to Σ_seq(ω_qrz')
```

Lemma 4.16. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, and $\vartheta = del$. Then, $t_{st \mapsto qrz}^{\Sigma_{ass}}(\sigma_{ass}^{del}(\omega_{st}^{\varphi}))$ translates $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$ to a sequence $\Sigma_{seq}(\omega_{qrz'})$ of $\sigma_{ass}^{del}(\omega_{qrz'})$ and $\sigma_{pause}(\omega_{qrz'})$ as specified in Definition 4.8. $\Sigma_{seq}(\omega_{qrz'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{qrz})$, preserves the SOS transition rules of $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$, and respects the single assignment per* macro step *constraint in* Quartz.

Proof The validity of Lemma 4.16 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{qrz'})$ with the syntax rules of $\Sigma_{seq}(\omega_{qrz})$ as specified in Section 3.3, where $\Sigma_{seq}(\omega_{qrz'}) = \{\sigma_{ass}^{del}(\omega_{qrz'}); \sigma_{pause}(\omega_{qrz'})\}$, which conforms to the syntax rules of $\sigma_{ass}^{del}(\omega_{qrz})$ and $\sigma_{pause}(\omega_{qrz})$. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{ass}^{del}(\omega_{qrz})$ in combination with $\sigma_{pause}(\omega_{qrz})$ with the SOS transition rules of $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$ (see Section 3.2 and 3.3). The semantics are preserved, considering that a PLC cycle can contain several *macro steps*. The single assignment per *macro step* is implicitly respected by using the delayed assignment operator of *Quartz* followed by a `pause` statement.

Illustrative Example for Lemma 4.16

As an example, the delayed assigned variable of the `ST_ASS_DEL` model is translated as follows[18]:

[17] Both models, ST and *Quartz*, are included in Appendix B.15, B.16, C.15, and C.16.
[18] Both models, ST and *Quartz*, are included in Appendix B.11 and C.11.

```
1    y0 := y0 + x0;
2        ⇒ next(y0) = y0 + x0;        // σ^del_ass(ω_qrz') added to Σ_seq(ω_qrz')
3        ⇒ pause;                     // σ_pause(ω_qrz') added to Σ_seq(ω_qrz')
```

4.2.9. Conditions

This step covers the translation function for translating conditions in ST models $\Sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st})$ to conditions in *Quartz* models $\Sigma^{\vartheta}_{cond}(\omega_{qrz'})$.

Definition 4.9 (Translation of conditions – ST-to-Quartz). *Let $\Omega^{\varphi}_{st} = \{\omega^{\varphi}_{st} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. A condition in ST models $\sigma_{cond}(\omega^{\varphi}_{st}) \in \Sigma_{cond}(\omega^{\varphi}_{st})$ is translated to a condition in* Quartz *models $\sigma_{cond}(\omega_{qrz'}) \in \Sigma_{cond}(\omega_{qrz})$ using the translation function $t^{\Sigma_{cond}}_{st\mapsto qrz}(\sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st}))$, which is described by Algorithm 7.*

Algorithm 7 Translate condition – ST-to-Quartz

Input: $\sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st})$
Output: $\sigma^{\vartheta}_{cond}(\omega_{qrz'})$
Translation Function $t^{\Sigma_{cond}}_{st\mapsto qrz}(\sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st}))$**:**

switch ϑ **do**
  case *it* **do**

$$\sigma^{it}_{cond}(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} \texttt{if}(t^{\tau}_{st\mapsto qrz}(\lambda^{b}(\sigma^{it}_{cond}(\omega^{\varphi}_{st})))\{ \\ \quad t^{\Sigma}_{st\mapsto qrz}(\Sigma_1(\sigma^{it}_{cond}(\omega^{\varphi}_{st})) \\ \} \end{array} \right\}$$

  end
  case ϑ = *ite* **do**

$$\sigma^{ite}_{cond}(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} \texttt{if}(t^{\tau}_{st\mapsto qrz}(\lambda^{b}(\sigma^{ite}_{cond}(\omega^{\varphi}_{st})))\{ \\ \quad t^{\Sigma}_{st\mapsto qrz}(\Sigma_1(\sigma^{ite}_{cond}(\omega^{\varphi}_{st})) \\ \}\texttt{else}\{ \\ \quad t^{\Sigma}_{st\mapsto qrz}(\Sigma_2(\sigma^{ite}_{cond}(\omega^{\varphi}_{st}))) \\ \} \end{array} \right\}$$

  end
end

Correctness

To check the correctness of Definition 4.9, the following lemma is used.

Lemma 4.17. *Let ω^{φ}_{st} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t^{\Sigma_{cond}}_{st\mapsto qrz}(\sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st}))$ translates $\sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st})$ to $\sigma^{\vartheta}_{cond}(\omega_{qrz'})$ as specified in Definition 4.9. $\sigma^{\vartheta}_{cond}(\omega_{qrz'})$ conforms to the syntax rules of $\sigma^{\vartheta}_{cond}(\omega_{qrz})$ and preserves the SOS transition rules of $\sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st})$.*

Proof The validity of Lemma 4.17 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma^{\vartheta}_{cond}(\omega_{qrz'})$ with the syntax rules of $\sigma^{\vartheta}_{cond}(\omega_{qrz})$ as specified in Section 3.3. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma^{\vartheta}_{cond}(\omega_{qrz})$ in Section 3.3 with the SOS transition rules of $\sigma^{\vartheta}_{cond}(\omega^{\varphi}_{st})$ in Section 3.2, considering that statements are assumed to terminate within the same PLC cycle.

Illustrative Example for Lemma 4.17

As an example, the conditions of the `ST_COND` model are translated as follows[19]:

```
1     IF x1 THEN                // initial condition
2         x2 := TRUE;
3     END_IF;
4         ⇒ if(x1){             // resulting condition
5         ⇒   x2 = true;
6         ⇒ }
7     ------------------------
8     IF x1 THEN                // initial condition
9         x0 := TRUE;
10    ELSE
11        x0 := FALSE;
12    END_IF;
13        ⇒ if(x1){             // resulting condition
14        ⇒   x0 = true;
15        ⇒ }else{
16        ⇒   x0  false;
17        ⇒ }
```

4.2.10. Loops

This step covers the translation function for translating loops in ST models $\Sigma^{\vartheta}_{loop}(\omega^{\varphi}_{st})$ to loops in *Quartz* models $\Sigma^{\vartheta}_{loop}(\omega_{qrz'})$.

Definition 4.10 (Translation of loops – ST-to-Quartz). *Let $\Omega^{\varphi}_{st} = \{\omega^{\varphi}_{st} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. A loop in ST models $\sigma^{\vartheta}_{loop}(\omega^{\varphi}_{st}) \in \Sigma_{loop}(\omega^{\varphi}_{st})$ is translated to a loop in* Quartz *models $\sigma^{\vartheta}_{loop}(\omega_{qrz'}) \in \Sigma_{loop}(\omega_{qrz'})$ using the translation function $t^{\Sigma_{loop}}_{st\mapsto qrz}(\sigma^{\vartheta}_{loop}(\omega^{\varphi}_{st}))$, which is described by Algorithm 8.*

Correctness

To check the correctness of Definition 4.10, the following lemmas are used.

Lemma 4.18. *Let ω^{φ}_{st} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t^{\Sigma_{loop}}_{st\mapsto qrz}(\sigma^{head}_{loop}(\omega^{\varphi}_{st}))$ translates $\sigma^{head}_{loop}(\omega^{\varphi}_{st})$ to $\sigma^{head}_{loop}(\omega_{qrz'})$ as specified in Definition 4.10. $\sigma^{head}_{loop}(\omega_{qrz'})$ conforms to the syntax rules of $\sigma^{head}_{loop}(\omega_{qrz})$ and preserves the SOS transition rules of $\sigma^{head}_{loop}(\omega^{\varphi}_{st})$.*

[19] Both models, ST and *Quartz*, are included in Appendix B.8 and C.8.

Algorithm 8 Translate loop – ST-to-Quartz

Input: $\sigma^{\vartheta}_{loop}(\omega^{\varphi}_{st})$

Output: $\sigma^{\vartheta}_{loop}(\omega_{qrz'})$

Translation Function $t^{\Sigma_{loop}}_{st\mapsto qrz}(\sigma^{\vartheta}_{loop}(\omega^{\varphi}_{st}))$**:**

switch ϑ **do**

case *head* **do**

$$\sigma^{head}_{loop}(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} \texttt{while(}t^{\tau}_{st\mapsto qrz}(\lambda^{b}(\sigma^{head}_{loop}(\omega^{\varphi}_{st})))\texttt{)\{} \\ \quad t^{\Sigma}_{st\mapsto qrz}(\Sigma(\sigma^{head}_{loop}(\omega^{\varphi}_{st})) \\ \quad \texttt{[pause;]} \\ \texttt{\}} \end{array} \right\}$$

▷ pause *is the last statement of an iteration that can be contained in* $t^{\Sigma}_{st\mapsto qrz}(\Sigma(\sigma^{head}_{loop}(\omega^{\varphi}_{st}))$

end

case $foot$ **do**

$$\sigma^{foot}_{loop}(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} \texttt{do\{} \\ \quad t^{\Sigma}_{st\mapsto qrz}(\Sigma(\sigma^{foot}_{loop}(\omega^{\varphi}_{st})) \\ \quad \texttt{[pause;]} \\ \texttt{\}while(!(}t^{\tau}_{st\mapsto qrz}(\lambda^{b}(\sigma^{foot}_{loop}(\omega^{\varphi}_{st})))\texttt{));} \end{array} \right\}$$

▷ pause *is the last statement of an iteration that can be contained in* $t^{\Sigma}_{st\mapsto qrz}(\Sigma(\sigma^{foot}_{loop}(\omega^{\varphi}_{st}))$

end

end

Proof The validity of Lemma 4.18 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma^{head}_{loop}(\omega_{qrz'})$ (including optional $\sigma_{pause}(\omega_{qrz'})$) with the syntax rules of $\sigma^{head}_{loop}(\omega_{qrz})$ and $\sigma_{pause}(\omega_{qrz})$ as specified in Section 3.3. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma^{head}_{loop}(\omega_{qrz})$ and optional $\sigma_{pause}(\omega_{qrz})$ in Section 3.3 with the SOS transition rules of $\sigma^{head}_{loop}(\omega^{\varphi}_{st})$ in Section 3.2, noting that $\sigma_{pause}(\omega_{qrz})$ is not always necessary (like for delayed assignments).

Illustrative Example for Lemma 4.18

As an example, the head-controlled loop of the `ST_LOOP_HEAD` model is translated as follows[20]:

```
1    i := i0;                  // initial loop
2    WHILE i <= i1 DO
3        y := i;
4        i := i + x2;
5    END_WHILE;
6    y := x1;
7        ⇒ i = i0;             // resulting loop
```

[20] Both models, ST and *Quartz*, are included in Appendix B.14 and C.14.

```
8         ⇒ while(i <= i1){
9         ⇒    y = i;
10        ⇒    next(i) = i + x2;
11        ⇒    pause;
12        ⇒ }
13        ⇒ y = x1;
```

Lemma 4.19. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t_{st \mapsto qrz}^{\Sigma_{loop}}(\sigma_{loop}^{foot}(\omega_{st}^{\varphi}))$ translates $\sigma_{loop}^{foot}(\omega_{st}^{\varphi})$ to $\sigma_{loop}^{foot}(\omega_{qrz'})$ as specified in Definition 4.10. $\sigma_{loop}^{foot}(\omega_{qrz'})$ conforms to the syntax rules of $\sigma_{loop}^{foot}(\omega_{qrz})$ and preserves the SOS transition rules of $\sigma_{loop}^{foot}(\omega_{st}^{\varphi})$.*

Proof The validity of Lemma 4.19 is proved as follows: First, the syntactic correctness is proved by comparing the resulting syntax of $\sigma_{loop}^{foot}(\omega_{qrz'})$ (including optional $\sigma_{pause}(\omega_{qrz'})$) with the syntax rules of $\sigma_{loop}^{foot}(\omega_{qrz})$ and $\sigma_{pause}(\omega_{qrz})$ as specified in Section 3.3. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{loop}^{foot}(\omega_{qrz})$ and optional $\sigma_{pause}(\omega_{qrz})$ in Section 3.3 with the SOS transition rules of $\sigma_{loop}^{foot}(\omega_{st}^{\varphi})$ in Section 3.2, noting that $\sigma_{pause}(\omega_{qrz})$ is not always necessary, such as for delayed assignments.

Illustrative Example for Lemma 4.19

As an example, the foot-controlled loop of the `ST_LOOP_FOOT` model is translated as follows[21]:

```
1     i := i0;                  // initial loop
2     REPEAT
3         y := x0;
4         i := i + x2;
5     UNTIL i>i1
6     END_REPEAT;
7     y := x1;
8         ⇒ i = i0;             // resulting loop
9         ⇒ do{
10        ⇒    y = x0;
11        ⇒    next(i) = i + x2;
12        ⇒    pause;
13        ⇒ }while(!(i > i1));
14        ⇒ y = x1;
```

[21] Both models, ST and *Quartz*, are included in Appendix B.13 and C.13.

4.2.11. Sequences

This step covers the translation function for translating sequences in ST models $\Sigma_{seq}(\omega_{st}^{\varphi})$ to sequences in *Quartz* models $\Sigma_{seq}(\omega_{qrz'})$.

Definition 4.11 (Translation of sequences – ST-to-Quartz). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements. There are the following variants of ST statements:*

- *POU invocations:* $\sigma_{inv}(\omega_{st}^{\varphi}) \in \Sigma_{inv}(\omega_{st}^{\varphi})$
- *Assignments:* $\sigma_{ass}^{imm}(\omega_{st}^{\varphi}) \in \Sigma_{ass}^{imm}(\omega_{st}^{\varphi})$, $\sigma_{ass}^{del}(\omega_{st}^{\varphi}) \in \Sigma_{ass}^{del}(\omega_{st}^{\varphi})$
- *Conditions:* $\sigma_{cond}^{head}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{head}(\omega_{st}^{\varphi})$, $\sigma_{cond}^{foot}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{foot}(\omega_{st}^{\varphi})$
- *Loops:* $\sigma_{loop}^{head}(\omega_{st}^{\varphi}) \in \Sigma_{loop}^{head}(\omega_{st}^{\varphi})$, $\sigma_{loop}^{foot}(\omega_{st}^{\varphi}) \in \Sigma_{loop}^{foot}(\omega_{st}^{\varphi})$

According to the Definition 4.7, 4.8, 4.9, and 4.10, these ST statements are translated to the following variants of Quartz *statements:*

- *Pause:* $\sigma_{pause}(\omega_{qrz'}) \in \Sigma_{pause}(\omega_{qrz'})$
- *Model invocations:* $\sigma_{inv}(\omega_{qrz'}) \in \Sigma_{inv}(\omega_{qrz'})$
- *Assignments:* $\sigma_{ass}^{imm}(\omega_{qrz'}) \in \Sigma_{ass}^{imm}(\omega_{qrz'})$, $\sigma_{ass}^{del}(\omega_{qrz'}) \in \Sigma_{ass}^{del}(\omega_{qrz'})$
- *Conditions:* $\sigma_{cond}^{head}(\omega_{qrz'}) \in \Sigma_{cond}^{head}(\omega_{qrz'})$, $\sigma_{cond}^{foot}(\omega_{qrz'}) \in \Sigma_{cond}^{foot}(\omega_{qrz'})$
- *Loops:* $\sigma_{loop}^{head}(\omega_{qrz'}) \in \Sigma_{loop}^{head}(\omega_{qrz'})$, $\sigma_{loop}^{foot}(\omega_{qrz'}) \in \Sigma_{loop}^{foot}(\omega_{qrz'})$
- *Sequences:* $\Sigma_{seq}(\omega_{qrz'})$

A set of the resulting Quartz *statements represents a sequence, denoted as* $\Sigma_{seq}(\omega_{qrz'})$*, which was introduced and appended in isolated transformation steps, such as for delayed assignments* $\Sigma_{seq}(\omega_{qrz'}) = \{\sigma_{seq}^{del}(\omega_{qrz'}); \sigma_{pause}(\omega_{qrz'})\}$ *in Definition 4.8. The translation function* $t_{st\mapsto qrz}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{qrz'}))$ *inserts* $\sigma_i(\omega_{qrz'}) \in \Sigma_{seq}(\omega_{qrz'})$ *to* $\omega_{qrz'}$*, following the process described by Algorithm 9.*

Algorithm 9 Add sequence – ST-to-Quartz

Input: $\Sigma_{seq}(\omega_{qrz'})$
Output: $\omega_{qrz'}$
Translation Function $t_{st\mapsto qrz}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{qrz'}))$**:**

```
forall σ_i ∈ Σ_seq(ω_qrz') do
    ω_qrz' ← add σ_i to the position w.r.t. its execution order and dependent
      constructs;
end
```

Correctness

To check the correctness of Definition 4.11, the following lemma is used.

Lemma 4.20. *Let ω_{st}^{φ} be translated to $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, $\Sigma_{seq}(\omega_{qrz'}) \neq \varnothing$, and $\Sigma_{seq}(\omega_{qrz'})$ be syntactically correct. Then, $t_{st \mapsto qrz}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{qrz'}))$ inserts each translated statement $\sigma_i \in \Sigma_{seq}(\omega_{qrz'})$ to $\omega_{qrz'}$. The resulting* Quartz *model $\omega_{qrz'}$ conforms to the syntax rules of ω_{qrz}, preserves the SOS transition rules of ω_{st}^{φ} (in particular with regard to the execution order of the statements), and respects the single assignment per* macro step *constraint in* Quartz.

Proof The validity of Lemma 4.20 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{qrz'})$ with the syntax rules of $\Sigma_{seq}(\omega_{qrz})$ as specified in Section 3.3 using induction on the number of statements to be added $\Sigma_{seq}(\omega_{qrz'})$:

1. **Base Case:** When $\Sigma_{seq}(\omega_{qrz'}) = \varnothing$, there are no statements to be added, which conforms to the syntax rules of $\delta_{\omega}(\omega_{qrz})$.
2. **Induction Hypothesis:** The lemma holds for any set of statements to be added $\Sigma_{seq}(\omega_{qrz'})$.
3. **Inductive Step:** Adding a statement results in a statement to be added that conforms to the syntax rules $\Sigma_{seq}(\omega_{qrz})$, because the syntactic correctness of this statement to be added has been proved in the corresponding section.

Second, the SOS transition rules and the single assignment per *macro step* constraint in *Quartz* are respected by the individual statements themselves. The order is respected by the order within the resulting *Quartz* model.

Illustrative Example for Lemma 4.20

As an example, below are the inserted statements in the resulting *Quartz* model of the `ST_SIMPLE_PRG1` model:

```
1   ...
2   loop{
3       immediate await(EI);
4
5       emit(DEBOUNCE_01_EI);                  // σ1 ∈ Σinv(ωqrz')
6       immediate await(DEBOUNCE_01_EO);       // σ2 ∈ Σinv(ωqrz')
7       PRG_OUT1 = DEBOUNCE_01_OUT;            // σ3 ∈ Σass(ωqrz')
8       PRG_ET_OFF = DEBOUNCE_01_ET_OFF;       // σ4 ∈ Σass(ωqrz')
9       ST_SIMPLE_FUN( ... );                  // σ5 ∈ Σinv(ωqrz')
10      PRG_OUT2 = ST_SIMPLE_FUN_11;           // σ6 ∈ Σass(ωqrz')
11
12      emit(EO);
13      pause;
14  }
15  || DEBOUNCE_01:DEBOUNCE( ... );            // σ1 ∈ Σinv(ωqrz')
```

4.3. Experimental Results

The applicability of the introduced translation functions is evaluated with the ST models listed in Table 4.1. The examples are listed with the *Source Lines of Code* (SLOC) metric [BM14] of the ST model and the experimental results. For evaluation purposes, the listed ST models and the expected *Quartz* models are manually implemented to verify the applicability of the isolated translation functions. To ensure the correctness of both models, ST and *Quartz*, they are compiled with the built-in compilers of *CODESYS* and *Averest*. The translation functions have been implemented as a prototype in *PLCreX*, resulting in the overall test strategy shown in Figure 4.6. The correctness of the resulting *Quartz* models is verified in two ways: (1) through manual reviews, differences between the expected *Quartz* models and the automatically generated models are identified, and (2) using the built-in compilers of *Averest*, the syntactic correctness of the automatically generated *Quartz* models is ensured. Both

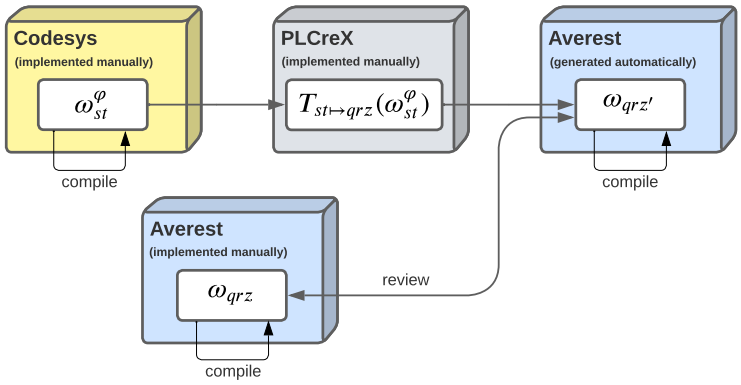


Figure 4.6.: *Test strategy to evaluate the ST-to-Quartz transformation*

tests passed for all examples (ignoring minor formatting differences between manually implemented and automatically generated *Quartz* models), with the following warnings:

- **Initializing input variables**: According to Lemma 4.2, input variables of *Quartz* models are always invoked with connected values, because input variables have only read access as specified in Section 3.3. Thus, input variables cannot be set to specific values, which is why the example *simple calculation* throws a warning for the initialized input variable, since the initialization is skipped during translation.

Based on the experimental results in Table 4.1, it can be concluded that the introduced translation functions are applicable and lead to correct *Quartz* models. They can be reused in model-based design, if a few conditions are considered. These are summarized in the following section.

Table 4.1.: *Set of ST models and test results to evaluate the applicability of the introduced ST-to-Quartz transformation*

Model	SLOC	Source	ω_{st}^{φ}	$\omega_{qrz'}$	Result
2-of-3 logic function	11	[Sch19]	B.1	C.1	passed
Alarm function	8	[Sch19]	B.2	C.2	passed
Analog value processing 1	8	[Sch19]	B.3	C.3	passed
Analog value processing 2	7	[Sch19]	B.4	C.4	passed
Arithmetic operators	20	self	B.5	C.5	passed
Boolean operators	14	self	B.6	C.6	passed
Compensation system	28	[Sch19]	B.7	C.7	passed
Condition statements	15	self	B.8	C.8	passed
Data types and fields	28	self	B.9	C.9	passed
Debounce	20	[GDV14]	B.10	C.10	passed
Delayed assignments	7	self	B.11	C.11	passed
Equality, inequality	9	self	B.12	C.12	passed
Foot-controlled loop	20	self	B.13	C.13	passed
Head-controlled loop	18	self	B.14	C.14	passed
Immediate assignments 1	13	self	B.15	C.15	passed
Immediate assignments 2	19	self	B.16	C.16	passed
Immediate assignments 3	10	self	B.17	C.17	passed
Invoke function 1	7	self	B.18	C.18	passed
Left detection	7	[Sch19]	B.19	C.19	passed
Numeric relations	11	self	B.20	C.20	passed
Off delay timer	39	[GDV14]	B.21	C.21	passed
On delay timer	39	[GDV14]	B.22	C.22	passed
RS-Flip-Flop	14	[GDV14]	B.23	C.23	passed
Right detection	7	[Sch19]	B.24	C.24	passed
SR-Flip-Flop	14	[GDV14]	B.25	C.25	passed
Simple calculation	16	[GDV14]	B.26	C.26	passed with warnings
Simple program	19	[GDV14]	B.27	C.27	passed
Tank control	19	[Sch19]	B.28	C.28	passed
Track correction	23	[Sch19]	B.29	C.29	passed
Two-Point controller	21	[Sch19]	B.30	C.30	passed

4.4. Summary

This chapter introduced the transformation of ST models to *Quartz* models. For this purpose, individual translation functions were defined that take into account the sequential execution order of ST statements. The applicability of these translation functions was demonstrated using a set of ST examples. Based on the presented lemmas and experimental results, the following theorem encapsulates the entire transformation:

Theorem 4.1 (ST-to-Quartz Translation). *Let $\omega_{st}^{\varphi} \in \Omega_{st}^{\varphi}$ be an ST model of variant $\varphi \in \{fb, fun, prg\}$, and let $T_{st \mapsto qrz}(\omega_{st}^{\varphi})$ be the model transformation of ω_{st}^{φ} to $\omega_{qrz'}$ using the translation functions defined in this chapter. Then, the resulting* Quartz *model $\omega_{qrz'}$:*

1. *Conforms to the syntax rules of ω_{qrz}*
2. *Preserves the semantics of ω_{st}^{φ}*
3. *Contains constructs corresponding to the constructs of ω_{st}^{φ} and preserves the intended functionality of ω_{st}^{φ} under the following conditions:*
   - $\varphi \in \{fb, fun, prg\}$
   - $\Delta_{idcl}(\omega_{st}^{\varphi}) = \Delta_{in}(\omega_{st}^{\varphi}) \cup \Delta_{out}(\omega_{st}^{\varphi}) \cup \Delta_{inout}(\omega_{st}^{\varphi})$, *where models are always invoked with defined input values, so no initializations are required for* $\Delta_{in}(\omega_{st}^{\varphi})$
   - $\Delta_{vdcl}(\omega_{st}^{\varphi}) = \Delta_{local}(\omega_{st}^{\varphi})$, *where variables are represented as memorized variables*
   - *Imported models are available as* Quartz *models*
   - *Functions are called formally complete in ST models*
   - $\forall \alpha^{[+]}(\omega_{st}^{\varphi}) : \alpha^{[+]} \in \{\alpha_{bv}^{bool}, \alpha_{bv}^{byte}, \alpha_{bv}^{word}, \alpha_{i}^{int}, \alpha_{i}^{dint}, \alpha_{i}^{uint}, \alpha_{i}^{udint}, \alpha_{dur}, \alpha^{+}\}$, *where α_{dur} can be treated as an unbounded integer and is specified in milliseconds*
   - $\forall \tau(\omega_{st}^{\varphi}) : \tau \in \{\tau_{misc}^{cst}, \tau_{misc}^{id}, \tau_{misc}^{\pi,\eta,\lambda}, \tau_{misc}^{br}, \tau_{misc}^{true}, \tau_{misc}^{false}, \tau_{misc}^{arr}, \tau_{misc}^{inv}, \tau_{comp}^{eq}, \tau_{comp}^{ne}, \tau_{comp}^{gt}, \tau_{comp}^{ge}, \tau_{comp}^{lt}, \tau_{comp}^{le}, \tau_{arith}^{mul}, \tau_{arith}^{div}, \tau_{arith}^{add}, \tau_{arith}^{sub}, \tau_{arith}^{expt}, \tau_{arith}^{mod}, \tau_{arith}^{um}, \tau_{cond}^{sel}\}$
   - $\forall \sigma(\omega_{st}^{\varphi}) : \sigma \in \{\sigma_{inv}^{\vartheta}, \sigma_{ass}^{\vartheta}, \sigma_{cond}^{\vartheta}, \sigma_{loop}^{\vartheta}\}$

Proof The validity of Theorem 4.1 is proved as follows:

1. **Syntax Conformance**: Lemma 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14 4.15, 4.16, 4.17, 4.18, 4.19, and 4.20 demonstrate that each translated construct conforms to the syntax rules of ω_{qrz} as specified in Section 3.3.
2. **Semantic Preservation**: The following lemmas address the preservation of semantics for their respective constructs.
   - Model declaration: Lemma 4.1
   - Interfaces: Lemma 4.2
   - Variables: Lemma 4.6
   - Data types and fields: Lemma 4.10

- POU imports: Lemma 4.11
- Expressions: Lemma 4.12
- POU invocations: Lemma 4.13 and 4.14
- Assignments: Lemma 4.15 and 4.16
- Conditions: Lemma 4.17
- Loops: Lemma 4.18 and 4.19
- Sequences: Lemma 4.20

3. **Construct Correspondence**: Given the conditions of the theorem, the provided definitions, proofs, and experimental results, it can be concluded that the translation functions produce corresponding constructs in ω_{qrz} for the considered constructs in ω_{st}^{φ}, preserving the original functionality.

Overall, this results in the following solutions to the challenges summarized in Section 4.1.

1. **Cyclic execution of *Quartz* models (with and without memory)**: The execution of *Quartz* models mimicking both POU variants, with and without memory, depends on the invocation by the external *Quartz* model, i.e., the external model is responsible for the cyclic execution.
2. **Event-driven execution of synchronous parallel threads**: An iteration of *Quartz* models mimicking POUs with memory is triggered via event input `EI` and returned via event output `EO` after a finite number of *macro steps* $n \geq 0$, thus realizing an external event-driven execution control of the resulting *Quartz* model.
3. **Sequential execution of synchronous parallel threads**: Due to the event-driven execution control realized with the event interfaces `EI` and `EO`, the invoking *Quartz* model can invoke instances depending on their feedback via `EO`, thus allowing the sequential execution of synchronous parallel threads taking into account an individual number of *macro steps* of the instances.
4. **Dynamic system time**: The global clock synchronization is realized by an additional memorized input `CLK`, i.e., the clock is controlled externally and processed within the *Quartz* model with read access.
5. **Translation of ST language constructs**: The solution follows from Theorem 4.1.

Chapter 5

Model Transformation of ST Models to SCL Models

Contents

The second approach to reusing existing POUs in model-based design is the transformation of ST models into SCL models, where the goal is to create a robust set of translation functions that ensure semantic preservation during the transition. In addition to the approaches presented in [WS24a; WS24b], it considers the following additional issues:

- **Model Declaration**: Mimicking the termination behavior of the initial model, i.e., distinguishing between models with and without memory
- **Interfaces and Variables**: Additional interfaces for event-driven execution control and an external controlled time (provided as a bounded integer)
- **Data Types and Fields**: Additional IEC 61131-3 data types

The correctness of the translation functions is proved by theoretical reasoning, which includes a detailed analysis of the resulting syntax and semantics compared to the syntax rules and semantics specified in Chapter 3. In addition, the theoretical results are evaluated with real-world and self-defined ST models.

This chapter is structured as follows: Section 5.1 introduces the high-level design flow and translation strategy. Section 5.2 defines the translation functions and theoretical analysis. Section 5.3 presents an evaluation of the theoretical results, and Section 5.4 summarizes the transformation.

5.1. High-Level Design Flow – ST-to-SCL

The high-level design flow for transforming an ST model $\omega_{st}^{\varphi} \in \Omega_{st}^{\varphi}$ of the variant *function block* ($\varphi = fb$), *function* ($\varphi = fun$), or *program* ($\varphi = prg$) to an SCL model $\omega_{scl} \in \Omega_{scl}$ is shown in Figure 5.1.

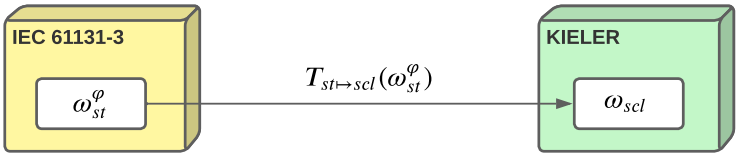


Figure 5.1.: *High-level design flow of the ST-to-SCL transformation*

The ST-to-SCL transformation $T_{st \mapsto scl}(\omega_{st}^{\varphi})$ includes the following transformation steps[1]:

1. $t_{st \mapsto scl}^{\delta_\omega}(\delta_\omega(\omega_{st}^{\varphi}))$: Model declaration (see Section 5.2.1)
2. $t_{st \mapsto scl}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$: Interfaces (see Section 5.2.2)
3. $t_{st \mapsto scl}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$: Variables (see Section 5.2.3)
4. $t_{st \mapsto scl}^{\alpha}(\alpha^{[+]}(\omega_{st}^{\varphi}))$: Data types and fields (see Section 5.2.4)
5. $t_{st \mapsto scl}^{\tau}(\tau(\omega_{st}^{\varphi}))$: Expressions (see Section 5.2.5)
6. $t_{st \mapsto scl}^{\Sigma_{ass}}(\sigma_{ass}^{\vartheta}(\omega_{st}^{\varphi}))$: Assignments (see Section 5.2.6)
7. $t_{st \mapsto scl}^{\Sigma_{cond}}(\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi}))$: Conditions (see Section 5.2.7)
8. $t_{st \mapsto scl}^{\Sigma_{loop}}(\sigma_{loop}^{\vartheta}(\omega_{st}^{\varphi}))$: Loops (see Section 5.2.8)
9. $t_{st \mapsto scl}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{scl'}))$: Sequences (see Section 5.2.9)

[1] It should be noted that based on experimental results in the latest version of *KIELER*, SCL models are not intended to import external SCL models (due to the minimal instruction set), and thus transformations of model imports and invocations are not considered in this chapter.

Translation Strategy:

This chapter introduces two translation strategies illustrated in Figure 5.2. In contrast to the strategies introduced in Chapter 4, in this approach both POU variants (with and without memory) run in an infinite loop. The difference is that the ST model is either transformed into an SCL model that (1) contains variables that are reset to their defaults at the beginning of each iteration (to mimic a POU without memory), or (2) runs in an infinite loop without resetting the variables (to mimic a POU with memory). Figure 5.2a shows the high-level runtime behavior of the resulting SCL model that follows the first translation strategy, where the set of input variables `IN` is read when an iteration of the SCL model is triggered via the input variable `EI` in *macro step* S_i. In the following *macro steps* during execution, the interfaces are synchronized, where only `CLK` may be updated externally and is read. Neither input variables `IN` are allowed to be updated nor output variables `OUT` are allowed to be processed externally until the current iteration is terminated. In *macro step* S_m, `OUT` is returned to the invoking model for external processing. The subsequent final *macro step* is used to switch to the next iteration (triggered in the next PLC cycle) and to reset `EO`. In contrast, Figure 5.2b shows the high-level runtime behavior of the resulting SCL model that follows the second translation strategy, where the model is initialized in the first *macro step* S_0 and then runs in an infinite loop until an iteration is triggered externally via the input variable `EI`. When an iteration is triggered in *macro step* S_i, the set of input variables `IN` and an external clock variable `CLK` are read and can be processed. In the following *macro steps* during execution, the interfaces are synchronized, where only `CLK` may be updated externally. Neither input variables `IN` are allowed to be updated nor output variables `OUT` are allowed to be processed externally until the iteration reaches the *macro step* S_m, where `EO` is set to `true`. In this *macro step*, `OUT` and `EO` are returned to the invoking model for external processing. The subsequent final *macro step* is used to switch to the next iteration (triggered in the next PLC cycle) and to reset `EO`.

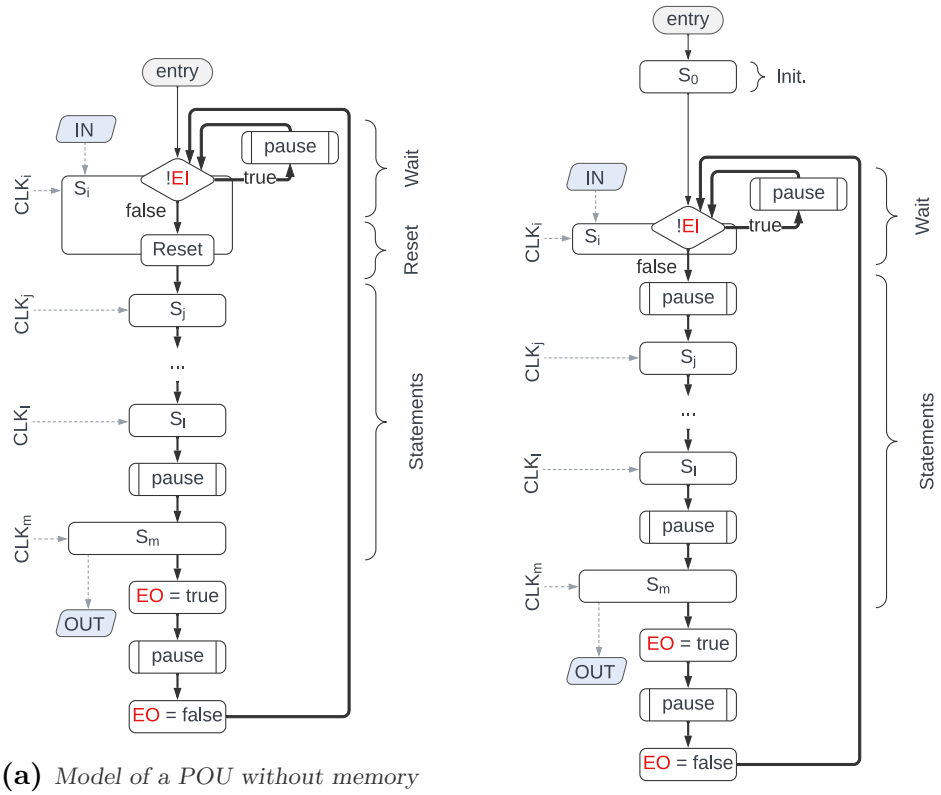
Challenges:

From this, the following challenges for translating ST models to SCL models can be derived:

1. Cyclic execution of SCL models (with and without memory)
2. Event-driven execution of SCL models
3. Dynamic system time
4. Translation of ST language constructs

5.2. From ST Models to SCL Models

This section defines the individual translation functions for translating an ST model $\omega_{st}^{\varphi} \in \Omega_{st}^{\varphi}$ to an SCL model $\omega_{scl'} \in \Omega_{scl}$ and analyzes the theoretical correctness.



(a) *Model of a POU without memory*

(b) *Model of a POU with memory*

Figure 5.2.: *ST-to-SCL translation strategies: high-level runtime behavior of the resulting SCL models*

5.2.1. Model Declaration

This step covers the translation function for translating an ST model declaration $\delta_\omega(\omega_{st}^\varphi)$ to an SCL model declaration $\delta_\omega(\omega_{scl'})$. According to the introduced translation strategies, the resulting SCL model of an ST model, variant $\varphi \in \{fb, fun, prg\}$, is executed in an infinite loop, reflecting the behavior of the original ST model with and without memory. For this, in each PLC cycle, the SCL model waits[2] until a loop iteration is triggered by an input, where the termination of the iteration is returned by an output.

Definition 5.1 (Model Declaration – ST-to-SCL). *Let* $\Omega_{st}^\varphi = \{\omega_{st}^\varphi \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements.* $\delta_\omega(\omega_{st}^\varphi)$ *is translated to* $\delta_\omega(\omega_{scl'})$ *using the translation function* $t_{st\mapsto scl}^{\delta_\omega}(\delta_\omega(\omega_{st}^\varphi))$*, which is described by Algorithm 10.*

Correctness

To check the correctness of Definition 5.1, the following lemma is used.

[2]The *wait* functionality is derived from the *immediate await* macro of *Quartz* [Sch09].

Algorithm 10 Translate model declaration – ST-to-SCL

Input: $\delta_\omega(\omega_{st}^\varphi)$
Output: $\delta_\omega(\omega_{scl'})$
Translation Function $t_{st\mapsto scl}^{\delta_\omega}(\delta_\omega(\omega_{st}^\varphi))$**:**

$$\delta_\omega(\omega_{scl'}) \leftarrow \left\{ \begin{array}{l} \texttt{module } a_n(\omega_{st}^\varphi)\{ \\ \quad t_{st\mapsto scl}^{\Delta_{idcl}}(\omega_{st}^\varphi) \quad t_{st\mapsto scl}^{\Delta_{vdcl}}(\omega_{st}^\varphi) \\ \quad \texttt{loop:} \\ \quad\quad \texttt{while(!EI)\{ pause; \}} \\ \quad\quad t_{st\mapsto scl}^{\Sigma}(\omega_{st}^\varphi) \\ \quad\quad \texttt{EO = true; pause; EO = false;} \\ \quad\quad \texttt{goto loop;} \\ \} \end{array} \right\}$$

Lemma 5.1. *Let ω_{st}^φ be translated to $\omega_{scl'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t_{st\mapsto scl}^{\delta_\omega}(\delta_\omega(\omega_{st}^\varphi))$ translates $\delta_\omega(\omega_{st}^\varphi)$ to $\delta_\omega(\omega_{scl'})$ as specified in Definition 5.1. $\delta_\omega(\omega_{scl'})$ conforms to the syntax rules of $\delta_\omega(\omega_{scl})$ and preserves the semantics of $\delta_\omega(\omega_{st}^\varphi)$ regarding its termination behavior.*

Proof The validity of Lemma 5.1 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\delta_\omega(\omega_{scl'})$ with the syntax rules of $\delta_\omega(\omega_{scl})$ as specified in Section 3.4. Second, the preservation of semantics is checked by comparing $[\![\delta_\omega(\omega_{scl})]\!]_\xi$ (see Section 3.4) with $[\![\delta_\omega(\omega_{st}^\varphi)]\!]_\xi$ for $\varphi \in \{fb, fun, prg\}$ (see Section 3.2). In particular, given the SOS transition rules of $\sigma_{loop}^{inf}(\omega_{scl'})$, $\sigma_{loop}^{head}(\omega_{scl'})$, $\sigma_{ass}^{imm}(\omega_{scl'})$, and $\sigma_{pause}(\omega_{scl'})$ as specified in Section 3.4, $\delta_\omega(\omega_{scl})$ results in an infinite loop whose iteration is executed immediately when input `EI` becomes true and triggers `EO` at the end of an iteration. Iterations are separated by $\sigma_{pause}(\omega_{scl})$. Consequently, $[\![\delta_\omega(\omega_{scl})]\!]_\xi$ preserves $[\![\delta_\omega(\omega_{st}^\varphi)]\!]_\xi$ regarding its termination behavior (see Section 3.2 and 3.4). The difference between POUs with and without memory is that interfaces and local variables are reset at the beginning of an iteration.

Illustrative Example for Lemma 5.1

Usage examples are shown below using snippets of the resulting SCL models[3] in combination with a visualization of the high-level runtime behavior. The resulting SCL model of example `ST_ALARM` in Figure 5.3 represents a model without memory, and the resulting SCL model of example `ST_ASS_DEL` in Figure 5.4 represents a model that preserves the state of the last *macro step* of the current PLC cycle for the next iteration (i.e., for the next PLC cycle).

[3] Both SCL models are included in Appendix D.2 and D.9.

```
1  module ST_ALARM{
2    input bool EI;
3    output bool EO;
4    input bool xSENSOR_L;
5    ...
6
7    loop:
8      while(!EI){ pause; }
9      ST_ALARM = false // Reset
10     ST_ALARM = ...
11     EO = true; pause; EO =
           false;
12     goto loop;
13 }
```

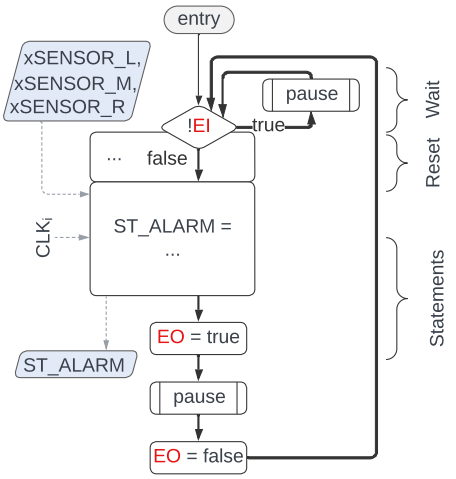


Figure 5.3.: *Resulting SCL model of example `ST_ALARM` (without memory)*

```
1  module ST_ASS_DEL{
2    input bool EI;
3    output bool EO;
4    int x0 = 2;
5    int y0 = 1;
6
7    loop:
8      while(!EI){ pause; }
9      y0 = y0 + x0;
10     EO = true; pause; EO =
           false;
11     goto loop;
12 }
```

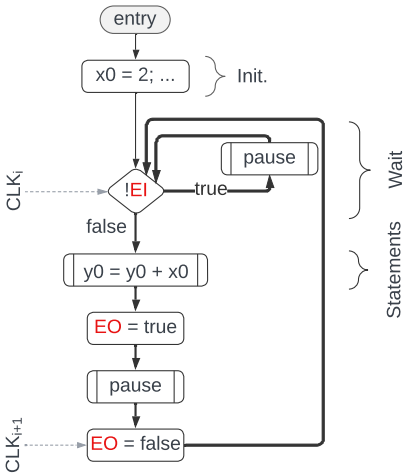


Figure 5.4.: *Resulting SCL model of example `ST_ASS_DEL` (with memory)*

5.2.2. Interfaces

This step covers the translation function for translating ST model interfaces $\Delta_{idcl}(\omega_{st}^{\varphi})$ to SCL model interfaces $\Delta_{idcl}(\omega_{scl'})$, where $\Delta_{idcl}(\omega_{st}^{\varphi}) = \Delta_{in}(\omega_{st}^{\varphi}) \cup \Delta_{out}(\omega_{st}^{\varphi}) \cup \Delta_{inout}(\omega_{st}^{\varphi})$.

Definition 5.2 (Interfaces – ST-to-SCL). *Let $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. $\Delta_{idcl}(\omega_{scl'})$ is derived from ω_{st}^{φ} and extended by interfaces for event-driven execution control and by an optional system time, using the translation function $t_{st \mapsto scl}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$, which is described by Algorithm 11.*

Correctness

To check the correctness of Definition 5.2, the following lemmas are used.

Algorithm 11 Translate interfaces – ST-to-SCL

Input: ω_{st}^{φ}
Output: $\Sigma_{seq}(\omega_{scl'})$, $\Delta_{in}(\omega_{scl'})$, $\Delta_{out}(\omega_{scl'})$, $\Delta_{inout}(\omega_{scl'})$,
Translation Function $t_{st\mapsto scl}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$**:**

- $\Delta_{in}(\omega_{scl'}) \leftarrow$ add `input bool EI;` ▷ *add Boolean input variable*
- $\Delta_{out}(\omega_{scl'}) \leftarrow$ add `output bool EO;` ▷ *add Boolean output variable*
- **if** ω_{st}^{φ} *contains time-based logic* **then**
  - $\Delta_{in}(\omega_{scl'}) \leftarrow$ add `input int CLK;` ▷ *add input variable*
- **end**
- **if** $\varphi = fun \wedge e_{rT} \neq \varnothing$, *where* $e_{rT} \in \mathcal{E}_{rT}(\omega_{st}^{\varphi}), \mathcal{E}_{rT}(\omega_{st}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{st}^{\varphi})$ **then**
  - $\Delta_{out}(\omega_{scl'}) \leftarrow$ add `output` $t_{st\mapsto scl}^{\alpha}(\alpha(e_{rT}))\ a_n(\omega_{st}^{\varphi})$; ▷ *add output variable*
- **end**
- **forall** $e_i \in \mathcal{E}_{i}(\omega_{st}^{\varphi})$ **do**
  - **if** $e_i = e_{iVs}$, *where* $e_{iVs} \in \mathcal{E}_{iVs}(\omega_{st}^{\varphi}), \mathcal{E}_{iVs}(\omega_{st}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{st}^{\varphi})$ **then**
    - $\Delta_{in}(\omega_{scl'}) \leftarrow$ add `input` $t_{st\mapsto scl}^{\alpha}(\alpha(e_i))\ a_n(e_i)$ `[=` $t_{st\mapsto scl}^{\tau_{misc}}(\pi)$`];` ▷ *add input variable with optional initialization*
  - **end**
  - **if** $e_i = e_{oVs}$, *where* $e_{oVs} \in \mathcal{E}_{oVs}(\omega_{st}^{\varphi}), \mathcal{E}_{oVs}(\omega_{st}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{st}^{\varphi})$ **then**
    - $\Delta_{out}(\omega_{scl'}) \leftarrow$ add `output` $t_{st\mapsto scl}^{\alpha}(\alpha(e_i))\ a_n(e_i)$ `[=` $t_{st\mapsto scl}^{\tau_{misc}}(\pi)$`];` ▷ *add output variable with optional initialization*
    - $\Sigma_{seq}(\omega_{scl'}) \leftarrow$ add $a_n(e_i)$ `=` $t_{st\mapsto scl}^{\tau_{misc}}(\pi)$; (if $\varphi = fun$) ▷ *add reset to default value if* $\varphi = fun$
  - **end**
  - **if** $e_i = e_{iOVs}$, *where* $e_{iOVs} \in \mathcal{E}_{iOVs}(\omega_{st}^{\varphi}), \mathcal{E}_{iOVs}(\omega_{st}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{st}^{\varphi})$ **then**
    - $\Delta_{inout}(\omega_{scl'}) \leftarrow$ add `input output` $t_{st\mapsto scl}^{\alpha}(\alpha(e_i))\ a_n(e_i)$ `[=` $t_{st\mapsto scl}^{\tau_{misc}}(\pi)$`];` ▷ *add inout variable with optional initialization*
  - **end**
- **end**

Lemma 5.2. *Let* ω_{st}^{φ} *be translated to* $\omega_{scl'}$ *and* $\varphi \in \{fb, fun, prg\}$. *Then, for each interface* $e_i \in \mathcal{E}_{i}(\omega_{st}^{\varphi})$, $t_{st\mapsto scl}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ *extends* $\Sigma_{seq}(\omega_{scl'})$, $\Delta_{in}(\omega_{scl'})$, $\Delta_{out}(\omega_{scl'})$, *or* $\Delta_{inout}(\omega_{scl'})$ *including optional initialization and reset to default value as specified in Definition 5.2.* $\Delta_{in}(\omega_{scl'})$, $\Delta_{out}(\omega_{scl'})$, *and* $\Delta_{inout}(\omega_{scl'})$ *conform to the syntax rules of* $\Delta_{in}(\omega_{scl})$, $\Delta_{out}(\omega_{scl})$, *and* $\Delta_{inout}(\omega_{scl})$ *regarding the storage class, data type, and name.* $\Sigma_{seq}(\omega_{scl'})$ *conforms to the syntax rules of* $\Sigma_{seq}(\omega_{scl})$. *With and without initialization,* $\Delta_{in}(\omega_{scl'})$, $\Delta_{out}(\omega_{scl'})$ *(in combination with* $\Sigma_{seq}(\omega_{scl'})$*), and* $\Delta_{inout}(\omega_{scl'})$ *preserves the semantics of* $\Delta_{in}(\omega_{st}^{\varphi})$, $\Delta_{out}(\omega_{st}^{\varphi})$, *and* $\Delta_{inout}(\omega_{st}^{\varphi})$ *regarding information flow, modifiability, and initialization.*

Proof The validity of Lemma 5.2 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{scl'})$, $\Delta_{in}(\omega_{scl'})$, $\Delta_{out}(\omega_{scl'})$, and $\Delta_{inout}(\omega_{scl'})$ with the syntax rules of $\Sigma_{seq}(\omega_{scl})$, $\Delta_{in}(\omega_{scl})$, $\Delta_{out}(\omega_{scl})$, and $\Delta_{inout}(\omega_{scl})$ as specified in Section 3.4 using induction on the number of added interfaces $\mathcal{E}_i(\omega_{st}^{\varphi})$, where $\varphi = fun$:

1. **Base Case:** When $\mathcal{E}_i(\omega_{st}^{\varphi}) = \varnothing$, there are no input variables, output variables, and inout variables to add, which trivially conforms to the syntax rules of $\Sigma_{seq}(\omega_{scl})$, $\Delta_{in}(\omega_{scl})$, $\Delta_{out}(\omega_{scl})$, and $\Delta_{inout}(\omega_{scl})$, since these sets remain unchanged and are optional.
2. **Induction Hypothesis:** The lemma holds for any set of input variables, output variables, and inout variables.
3. **Inductive Step:** Adding an element to input variables, output variables, and inout variables results in an additional element in $\Sigma_{seq}(\omega_{scl'})$, $\Delta_{in}(\omega_{scl'})$, $\Delta_{out}(\omega_{scl'})$, and $\Delta_{inout}(\omega_{scl'})$. Their syntax still conforms to the syntax rules of $\Sigma_{seq}(\omega_{scl})$, $\Delta_{in}(\omega_{scl})$, $\Delta_{out}(\omega_{scl})$, and $\Delta_{inout}(\omega_{scl})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{in}(\omega_{scl})]\!]_{\xi}$, $[\![\Delta_{out}(\omega_{scl})]\!]_{\xi}$ in combination with the SOS transition rules of $\Sigma_{seq}(\omega_{scl})$, and $[\![\Delta_{inout}(\omega_{scl})]\!]_{\xi}$ (see Section 3.4) with $[\![\Delta_{in}(\omega_{st}^{\varphi})]\!]_{\xi}$, $[\![\Delta_{out}(\omega_{st}^{\varphi})]\!]_{\xi}$, and $[\![\Delta_{inout}(\omega_{st}^{\varphi})]\!]_{\xi}$ (see Section 3.2).

Illustrative Example for Lemma 5.2

As an example, below are the derived interfaces of the `ST_TON` model[4], where one interface is initialized:

```
1    input bool IN;                    // added to Δin(ωscl')
2    input int PT;                     // added to Δin(ωscl')
3    output bool Q1 = false;           // added to Δout(ωscl')
4    output int ET;                    // added to Δout(ωscl')
```

Lemma 5.3. *Let ω_{st}^{φ} be translated to $\omega_{scl'}$, $\varphi \in \{fb, fun, prg\}$, and $\omega_{scl'}$ be event-driven. Then, $t_{st\mapsto scl}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ adds the additional input variable `EI` to $\Delta_{in}(\omega_{scl'})$ and the additional output variable `EO` to $\Delta_{out}(\omega_{scl'})$ as specified in Definition 5.2. $\Delta_{in}(\omega_{scl'})$ and $\Delta_{out}(\omega_{scl'})$ conform to the syntax rules of $\Delta_{in}(\omega_{scl})$ and $\Delta_{out}(\omega_{scl})$.*

Proof The validity of Lemma 5.3 is proved by comparing the resulting syntax of $\Delta_{in}(\omega_{scl'})$ and $\Delta_{out}(\omega_{scl'})$ with the syntax rules of $\Delta_{in}(\omega_{scl})$ and $\Delta_{out}(\omega_{scl})$ as specified in Section 3.4.

[4]Both models, ST and SCL, are included in Appendix B.22 and D.19.

Illustrative Example for Lemma 5.3

As an example, below are the derived interfaces of the ST_TON model[5]:

```
1    input bool EI;                    // added to Δin(ωscl′)
2    output bool EO;                   // added to Δout(ωscl′)
```

Lemma 5.4. *Let ω_{st}^{φ} be translated to $\omega_{scl'}$, $\varphi \in \{fb, fun, prg\}$, and ω_{st}^{φ} contains time-based logic, where time is a readable variable and is synchronized externally. Then, $t_{st\mapsto scl}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ adds an additional input to $\Delta_{in}(\omega_{scl'})$ as specified in Definition 5.2. $\Delta_{in}(\omega_{scl'})$ conforms to the syntax rules of $\Delta_{in}(\omega_{scl})$.*

Proof The validity of Lemma 5.4 is proved by comparing the resulting syntax of $\Delta_{in}(\omega_{scl'})$ with the syntax rules of $\Delta_{in}(\omega_{scl})$ as specified in Section 3.4.

Illustrative Example for Lemma 5.4

As an example, below is the derived input of the ST_TON model[6]:

```
1    input int CLK;                    // added to Δin(ωscl′)
```

Lemma 5.5. *Let ω_{st}^{fun} be translated to $\omega_{scl'}$ and ω_{st}^{fun} has a return value, which is processed by ω_{st}^{fun}. Then, $t_{st\mapsto scl}^{\Delta_{idcl}}(\omega_{st}^{\varphi})$ adds an additional output to $\Delta_{out}(\omega_{scl'})$ as specified in Definition 5.2. $\Delta_{out}(\omega_{scl'})$ conforms to the syntax rules of $\Delta_{out}(\omega_{scl})$.*

Proof The validity of Lemma 5.5 is proved by comparing the resulting syntax of $\Delta_{out}(\omega_{scl'})$ with the syntax rules of $\Delta_{out}(\omega_{scl})$ as specified in Section 3.4.

Illustrative Example for Lemma 5.5

As an example, below is the derived output of the ST_SIMPLE_FUN model[7]:

```
1    output float ST_SIMPLE_FUN;       // added to Δout(ωscl′)
```

[5] Both models, ST and SCL, are included in Appendix B.22 and D.19.
[6] Both models, ST and SCL, are included in Appendix B.22 and D.19.
[7] Both models, ST and SCL, are included in Appendix B.26 and D.23.

5.2.3. Variables

This step covers the translation function for translating local ST model variables $\Delta_{vdcl}(\omega_{st}^{\varphi})$ to local SCL model variables $\Delta_{vdcl}(\omega_{scl'})$, where $\Delta_{vdcl}(\omega_{st}^{\varphi}) = \Delta_{local}(\omega_{st}^{\varphi}) \cup \Delta_{inst}(\omega_{st}^{\varphi})$.

Definition 5.3 (Variables – ST-to-SCL). *Let $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. $\Delta_{vdcl}(\omega_{scl'})$ is derived from ω_{st}^{φ} using the translation function $t_{st\mapsto scl}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$, which is described by Algorithm 12.*

Algorithm 12 Translate variables – ST-to-SCL

Input: ω_{st}^{φ}
Output: $\Sigma_{seq}(\omega_{scl'})$, $\Delta_{local}(\omega_{scl'})$
Translation Function $t_{st\mapsto scl}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$**:**

```
forall e_lVs ∈ E_i(ω_st^φ) do
    if e_d(e_lVs) = ∅ then
        Δ_local(ω_scl') ← add t^α_{st↦scl}(α(e_lVs)) a_n(e_lVs) [= t^{τ_misc}_{st↦scl}(π)];
                                  ▷ add local variable with optional initialization
        Σ_seq(ω_scl') ← add a_n(e_lVs) = t^{τ_misc}_{st↦scl}(π); (if φ = fun )
                                  ▷ add reset to default value if φ = fun
    end
end
```

Correctness

To check the correctness of Definition 5.3, the following lemma is used.

Lemma 5.6. *Let ω_{st}^{φ} be translated to $\omega_{scl'}$ and $\varphi \in \{fb, fun, prg\}$. Then, for each local variable $e_{lVs} \in \mathcal{E}_i(\omega_{st}^{\varphi})$ that is not derived from external models $e_d(e_{lVs}) = \varnothing$, $t_{st\mapsto scl}^{\Delta_{vdcl}}(\omega_{st}^{\varphi})$ adds a local variable including optional initialization value to $\Delta_{local}(\omega_{scl'})$ and reset to default to $\Sigma_{seq}(\omega_{scl'})$ as specified in Definition 5.3. $\Delta_{local}(\omega_{scl'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{scl})$ regarding the storage class, data type, and name. $\Sigma_{seq}(\omega_{scl'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{scl})$. With and without initialization, $\Delta_{local}(\omega_{scl'})$ (in combination with $\Sigma_{seq}(\omega_{scl'})$) preserves the semantics of $\Delta_{local}(\omega_{st}^{\varphi})$ regarding modifiability and initialization.*

Proof The validity of Lemma 5.6 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{scl'})$ and $\Delta_{local}(\omega_{scl'})$ with the syntax rules of $\Sigma_{seq}(\omega_{scl})$ and $\Delta_{local}(\omega_{scl})$ as specified in Section 3.4 using induction on the number of added variables $\mathcal{E}_i(\omega_{st}^{\varphi})$, where $\varphi = fun$:

1. **Base Case:** When $\mathcal{E}_i(\omega_{st}^{\varphi}) = \varnothing$, there are no local variables to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{scl})$ and $\Sigma_{seq}(\omega_{scl})$, since these sets remain unchanged and are optional.
2. **Induction Hypothesis:** The lemma holds for any set of local variables with possible initialization.
3. **Inductive Step:** Adding an element to local variables with initialization results in an additional element in $\Delta_{local}(\omega_{scl'})$ and $\Sigma_{seq}(\omega_{scl'})$. Their syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{scl})$ and $\Sigma_{seq}(\omega_{scl})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{local}(\omega_{scl})]\!]_{\xi}$ in combination with the SOS transition rules of $\Sigma_{seq}(\omega_{scl})$ (see Section 3.4) with $[\![\Delta_{local}(\omega_{st}^{\varphi})]\!]_{\xi}$ (see Section 3.2).

Illustrative Example for Lemma 5.6

As an example, below are the derived local variables of the `ST_ASS_DEL` model[8], where both variables are initialized:

```
1    int x0 = 2;              // added to Δ_local(ω_scl')
2    int y0 = 1;              // added to Δ_local(ω_scl')
```

5.2.4. Data Types and Fields

This step covers the translation function for translating ST data types and fields $\mathcal{A}^{[+]}(\omega_{st}^{\varphi})$ to SCL data types and fields $\mathcal{A}^{[+]}(\omega_{scl'})$.

Definition 5.4 (Translation of FBD data types and fields). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements. A ST data type or field* $\alpha^{[+]}(\omega_{st}^{\varphi}) \in \mathcal{A}^{[+]}(\omega_{st}^{\varphi})$ *is translated to an SCL data type and field* $\alpha^{[+]}(\omega_{scl'}) \in \mathcal{A}^{[+]}(\omega_{scl'})$ *using the translation function* $t^{\alpha}_{st \mapsto scl}(\alpha^{[+]}(\omega_{st}^{\varphi}))$*, which is described by Algorithm 39 in Appendix J.0.1*[9]*.*

Correctness

To check the correctness of Definition 5.4, the following lemma is used, noting that the exclusion of data types is based on experimental tests in the latest version of *KIELER* with restriction to internal data types.

Lemma 5.7. *Let* ω_{st}^{φ} *be translated to* $\omega_{scl'}$*,* $\varphi \in \{fb, fun, prg\}$*, and* bit vector*,* integer*,* floating point*, and* duration *be the considered data type categories* $\mathcal{A}^{[+]}(\omega_{st}^{\varphi})$ *as specified in Section 3.2. Then,* $t^{\alpha}_{st \mapsto scl}(\alpha^{[+]}(\omega_{st}^{\varphi}))$ *translates* $\alpha^{[+]}(\omega_{st}^{\varphi})$ *to* $\alpha^{[+]}(\omega_{scl'})$ *as specified in Definition 5.4.* $\alpha^{[+]}(\omega_{scl'})$ *conforms to the syntax rules of* $\alpha^{[+]}(\omega_{scl})$ *and preserves the semantics of* $\alpha^{[+]}(\omega_{st}^{\varphi})$ *regarding boundaries, precision, resolution, and defaults (if applicable), with the following restrictions:*

[8] Both models, ST and SCL, are included in Appendix B.11 and D.9.

[9] This algorithm describes an intuitive mapping, which is moved to the appendix for the sake of readability, but is not necessary for understanding the following lemma.

- $\forall \alpha(\omega_{st}^{\varphi}) : \alpha \in \{\alpha_{bv}^{byte} \alpha_{bv}^{word}, \alpha_{i}^{udint}, \}$: *Data types are not supported by internal SCL data types*
- $\forall \alpha(\omega_{st}^{\varphi}) : \alpha(\omega_{st}^{\varphi}) \in \{\alpha_{dur}^{time}, \alpha_{i}^{int}, \alpha_{i}^{uint}, \alpha_{i}^{dint}\}$: *Boundaries are changed to those of* $\alpha_{i}^{int}(\omega_{scl})$ *(see Section 3.5)*

Proof The validity of Lemma 5.7 is proved as follows: First, the syntactic correctness is checked by comparing all resulting data types and fields $\alpha^{[+]}(\omega_{scl'})$ with syntax rules $\alpha^{[+]}(\omega_{scl})$ as specified in Section 3.4. Second, the semantic correctness is checked by comparing $[\![\alpha^{[+]}(\omega_{scl})]\!]_{\xi}$ with $[\![\alpha^{[+]}(\omega_{st}^{\varphi})]\!]_{\xi}$ (see Section 3.2 and 3.4), taking into account that the resolution of the duration data type is restricted to milliseconds.

Illustrative Example for Lemma 5.7

Usage examples are given by illustrative examples of previous lemmas, such as Lemma 5.6.

5.2.5. Expressions

This step covers the translation function for translating expressions in ST models $\mathcal{T}(\omega_{st}^{\varphi})$ to expressions in SCL models $\mathcal{T}(\omega_{scl'})$.

Definition 5.5 (Expressions – ST-to-SCL). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements. An expression in ST models* $\tau(\omega_{st}^{\varphi}) \in \mathcal{T}(\omega_{st}^{\varphi})$ *is translated to an expression in SCL models* $\tau(\omega_{scl'}) \in \mathcal{T}(\omega_{scl'})$ *using the translation function* $t_{st \mapsto scl}^{\tau}(\tau(\omega_{st}^{\varphi}))$*, which is described by Algorithm 40 in Appendix J.0.2*[10]*.*

Correctness

To check the correctness of Definition 5.5, the following lemma is used, noting that the exclusion of expressions is based on experimental tests in the latest version of *KIELER* with restriction to internal operators.

Lemma 5.8. *Let* ω_{st}^{φ} *be translated to* $\omega_{scl'}$*,* $\varphi \in \{fb, fun, prg\}$*, and* miscellaneous*,* compare operators*,* arithmetic operators*,* conditional operator*, and* boolean operators *be the considered expression categories* $\mathcal{T}(\omega_{st}^{\varphi})$ *as specified in Section 3.2. Then, for each* $\tau(\omega_{st}^{\varphi}) \in \mathcal{T}(\omega_{st}^{\varphi})$*,* $t_{st \mapsto scl}^{\tau}(\tau(\omega_{st}^{\varphi}))$ *translates* $\tau(\omega_{st}^{\varphi})$ *to* $\tau(\omega_{scl'})$ *as specified in Definition 5.5.* $\tau(\omega_{scl'})$ *conforms to the syntax rules of* $\tau(\omega_{scl})$ *and preserves the semantics of* $\tau(\omega_{st}^{\varphi})$ *regarding the type system and SOS rules, with the following restrictions:*

[10] This algorithm describes an intuitive mapping, which is moved to the appendix for the sake of readability, but is not necessary for understanding the following lemma.

- $\forall \tau(\omega_{st}^{\vartheta}) : \tau \notin \{\tau_{misc}^{inv}, \tau_{arith}^{expt}\}$, *because they are not covered by the internal SCL operators*

Proof The validity of Lemma 5.8 is proved as follows: First, the syntactic correctness is checked by comparing all resulting expressions $\tau(\omega_{scl'})$ with syntax rules $\tau(\omega_{scl})$ as specified in Section 3.4. Second, the semantic correctness is checked by comparing $[\![\tau(\omega_{scl})]\!]_{\xi}$ with $[\![\tau(\omega_{st}^{\varphi})]\!]_{\xi}$, their type system, and SOS transition rules as specified in Section 3.2 and 3.4.

Illustrative Example for Lemma 5.8

As an example, below are two derived expressions of the `ST_SIMPLE_FUN` model[11]:

```
1    (COUNT + 1)                    // result of t^τ_{st↦scl}(COUNT+1)
2    ((A1 * B1) / C1)               // result of t^τ_{st↦scl}((A1*B1)/C1)
```

5.2.6. Assignments

This step covers the translation function for translating assignments in ST models $\Sigma_{ass}^{\vartheta}(\omega_{st}^{\varphi})$ to assignments in SCL models $\Sigma_{ass}^{\vartheta}(\omega_{scl'})$.

Definition 5.6 (Translation of assignments – ST-to-SCL). *Let $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements and $\Sigma_{ass}^{\vartheta}(\omega_{st}^{\varphi})$ be the set of assigned variables. An immediate assignment $\sigma_{ass}^{imm}(\omega_{st}^{\varphi}) \in \Sigma_{ass}^{imm}(\omega_{st}^{\varphi})$ (i.e., $rhs(\sigma_{ass}^{imm}(\omega_{st}^{\varphi}))$ does not depend on $lhs(\sigma_{ass}^{imm}(\omega_{st}^{\varphi}))$ and a delayed assignment in ST models $\sigma_{ass}^{del}(\omega_{st}^{\varphi}) \in \Sigma_{ass}^{del}(\omega_{st}^{\varphi})$ (i.e., $rhs(\sigma_{ass}^{del}(\omega_{st}^{\varphi}))$ does depend on $lhs(\sigma_{ass}^{del}(\omega_{st}^{\varphi}))$ is translated to a corresponding immediate assignment $\sigma_{ass}^{imm}(\omega_{scl'}) \in \Sigma_{ass}^{imm}(\omega_{scl'})$ and delayed assignment $\sigma_{ass}^{del}(\omega_{scl'}) \in \Sigma_{ass}^{del}(\omega_{scl'})$ in SCL models using the translation function $t_{st\mapsto scl}^{\Sigma_{ass}}(\sigma_{ass}^{\vartheta}(\omega_{st}^{\varphi}))$, which is described by Algorithm 13.*

Correctness

To check the correctness of Definition 5.6, the following lemmas are used.

Lemma 5.9. *Let ω_{st}^{φ} be translated to $\omega_{scl'}$, $\varphi \in \{fb, fun, prg\}$, and $\vartheta = imm$. Then, $t_{st\mapsto scl}^{\Sigma_{ass}}(\sigma_{ass}^{imm}(\omega_{st}^{\varphi}))$ identifies the left-hand side and right-hand side of $\sigma_{ass}^{imm}(\omega_{st}^{\varphi})$, and thus translates $\sigma_{ass}^{imm}(\omega_{st}^{\varphi})$ to $\sigma_{ass}^{imm}(\omega_{scl'})$ as specified in Definition 5.6. $\sigma_{ass}^{imm}(\omega_{scl'})$ conforms to the syntax rules of $\sigma_{ass}^{imm}(\omega_{scl})$ preserves the SOS rules of $\sigma_{ass}^{imm}(\omega_{st}^{\varphi})$ and $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$, and respects the SCMoC.*

[11] Both models, ST and SCL, are included in Appendix B.26 and D.23.

Algorithm 13 Translate assignment – ST-to-SCL

Input: $\sigma_{ass}^{\vartheta}(\omega_{st}^{\varphi})$
Output: $\sigma_{ass}^{\vartheta}(\omega_{scl'})$
Translation Function $t_{st\mapsto scl}^{\Sigma_{ass}}(\sigma_{ass}^{\vartheta}(\omega_{st}^{\varphi}))$**:**

```
switch ϑ do
    case imm do
        σ^imm_ass(ω_scl') ← lhs(σ^imm_ass(ω^φ_st)) = t^τ_{st↦scl}(rhs(σ^imm_ass));
    end
    case del do
        σ^del_ass(ω_scl') ← lhs(σ^del_ass(ω^φ_st)) = t^τ_{st↦scl}(rhs(σ^del_ass));
    end
end
```

Proof The validity of Lemma 5.9 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma_{ass}^{imm}(\omega_{scl'})$ with the syntax rules of $\sigma_{ass}^{imm}(\omega_{scl})$ as specified in Section 3.4. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{ass}^{imm}(\omega_{scl})$ with the SOS transition rules of $\sigma_{ass}^{imm}(\omega_{st}^{\varphi})$ (see Section 3.2 and 3.4). The SCMoC is implicitly respected, as demonstrated by the SOS transition rules.

Illustrative Example for Lemma 5.9

As an example, the immediately assigned variables of the `ST_ASS_IMM1` model and `ST_ASS_IMM2` model are translated as follows[12]:

```
1  y:=x;
2      ⇒ y=x;              // resulting σ^imm_ass(ω_scl')
3  ------------------------
4  y0:=x0; y1:=x1;
5      ⇒ y0=x0;            // resulting σ^imm_ass(ω_scl')
6      ⇒ y1=x1;            // resulting σ^imm_ass(ω_scl')
7  ------------------------
8  y0:=x0; y1:=x1; y0:=x2;
9      ⇒ y0=x0;            // resulting σ^imm_ass(ω_scl')
10     ⇒ y1=x1;            // resulting σ^imm_ass(ω_scl')
11     ⇒ y0=x2;            // resulting σ^imm_ass(ω_scl')
12 ------------------------
13 y2:=x0; y2:=x1;
14     ⇒ y2=x0;            // resulting σ^imm_ass(ω_scl')
15     ⇒ y2=x1;            // resulting σ^imm_ass(ω_scl')
16 ------------------------
17 y0:=x0; x0:=y0+x1;
18     ⇒ y0=x0;            // resulting σ^imm_ass(ω_scl')
19     ⇒ x0=y0+x1;         // resulting σ^imm_ass(ω_scl')
```

Lemma 5.10. *Let ω_{st}^{φ} be translated to $\omega_{scl'}$, $\varphi \in \{fb, fun, prg\}$, and $\vartheta = del$. Then, $t_{st\mapsto scl}^{\Sigma_{ass}}(\sigma_{ass}^{del}(\omega_{st}^{\varphi}))$ identifies the left-hand side and right-hand side of $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$, and thus translates $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$ to $\sigma_{ass}^{del}(\omega_{scl'})$ as specified in Definition 5.6. $\sigma_{ass}^{del}(\omega_{scl'})$ conforms to the syntax rules of $\sigma_{ass}^{del}(\omega_{scl})$, preserves the SOS rules of $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$, and respects the SCMoC.*

[12] Both models, ST and SCL, are included in Appendix B.15, B.16, D.13, and D.14.

Proof The validity of Lemma 5.10 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma_{ass}^{del}(\omega_{scl'})$ with the syntax rules of $\sigma_{ass}^{del}(\omega_{scl})$ as specified in Section 3.4. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{ass}^{del}(\omega_{scl})$ with the SOS transition rules of $\sigma_{ass}^{del}(\omega_{st}^{\varphi})$ (see Section 3.2 and 3.4). The SCMoC is implicitly respected, as demonstrated by the SOS transition rules.

Illustrative Example for Lemma 5.10

As an example, the delayed assigned variable of the `ST_ASS_DEL` model is translated as follows[13]:

```
1    y0:=y0+x0;
2        ⇒ y0=y0+x0;          // resulting σ^del_ass(ω_scl')
```

5.2.7. Conditions

This step covers the translation function for translating conditions in ST models $\Sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi})$ to conditions in SCL models $\Sigma_{cond}^{\vartheta}(\omega_{scl'})$.

Definition 5.7 (Translation of conditions – ST-to-SCL). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements. A condition in ST models* $\sigma_{cond}(\omega_{st}^{\varphi}) \in \Sigma_{cond}(\omega_{st}^{\varphi})$ *is translated to a condition in SCL models* $\sigma_{cond}(\omega_{scl'}) \in \Sigma_{cond}(\omega_{scl'})$ *using the translation function* $t_{st\mapsto scl}^{\Sigma_{cond}}(\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi}))$*, which is described by Algorithm 14.*

Algorithm 14 Translate condition – ST-to-SCL

Input: $\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi})$
Output: $\sigma_{cond}^{\vartheta}(\omega_{scl'})$
Translation Function $t_{st\mapsto scl}^{\Sigma_{cond}}(\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi}))$**:**

switch ϑ **do**

case *it* **do**

$$\sigma_{cond}^{it}(\omega_{scl'}) \leftarrow \left\{ \begin{array}{l} \texttt{if(}t_{st\mapsto scl}^{\tau}(\lambda^{b}(\sigma_{cond}^{it}(\omega_{st}^{\varphi})))\texttt{\{} \\ \quad t_{st\mapsto scl}^{\Sigma}(\Sigma_1(\sigma_{cond}^{it}(\omega_{st}^{\varphi})) \\ \texttt{\}} \end{array} \right\}$$

end

case ϑ = *ite* **do**

$$\sigma_{cond}^{ite}(\omega_{scl'}) \leftarrow \left\{ \begin{array}{l} \texttt{if(}t_{st\mapsto scl}^{\tau}(\lambda^{b}(\sigma_{cond}^{ite}(\omega_{st}^{\varphi})))\texttt{\{} \\ \quad t_{st\mapsto scl}^{\Sigma}(\Sigma_1(\sigma_{cond}^{ite}(\omega_{st}^{\varphi})) \\ \texttt{\}else\{} \\ \quad t_{st\mapsto scl}^{\Sigma}(\Sigma_2(\sigma_{cond}^{ite}(\omega_{st}^{\varphi}))) \\ \texttt{\}} \end{array} \right\}$$

end

end

[13] Both models, ST and SCL, are included in Appendix B.11 and D.9.

Correctness

To check the correctness of Definition 5.7, the following lemma is used.

Lemma 5.11. *Let ω_{st}^{φ} be translated to $\omega_{scl'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t_{st \mapsto scl}^{\Sigma_{cond}}(\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi}))$ translates $\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi})$ to $\sigma_{cond}^{\vartheta}(\omega_{scl'})$ as specified in Definition 5.7. $\sigma_{cond}^{\vartheta}(\omega_{scl'})$ conforms to the syntax rules of $\sigma_{cond}^{\vartheta}(\omega_{scl})$ and preserves the SOS rules of $\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi})$.*

Proof The validity of Lemma 5.11 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma_{cond}^{\vartheta}(\omega_{scl'})$ with the syntax rules of $\sigma_{cond}^{\vartheta}(\omega_{scl})$ in Section 3.4. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{cond}^{\vartheta}(\omega_{scl})$ in Section 3.4 with the SOS transition rules of $\sigma_{cond}^{\vartheta}(\omega_{st}^{\varphi})$ in Section 3.2, considering that statements are assumed to terminate within the same PLC cycle.

Illustrative Example for Lemma 5.11

As an example, the conditions of the `ST_COND` model are translated as follows[14]:

```
1     IF x1 THEN                // initial condition
2         x2:=TRUE;
3     END_IF;
4         ⇒ if(x1){             // resulting condition
5         ⇒   x2=true;
6         ⇒ }
7     ------------------------
8     IF x1 THEN                // initial condition
9         x0:=TRUE;
10    ELSE
11        x0:= FALSE;
12    END_IF;
13        ⇒ if(x1){             // resulting condition
14        ⇒   x0=true;
15        ⇒ }else{
16        ⇒   x0=false;
17        ⇒ }
```

5.2.8. Loops

This step covers the translation function for translating loops in ST models $\Sigma_{loop}^{\vartheta}(\omega_{st}^{\varphi})$ to loops in SCL models $\Sigma_{loop}^{\vartheta}(\omega_{scl'})$.

Definition 5.8 (Translation of loops – ST-to-SCL). *Let $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible ST model elements. A loop in ST models $\sigma_{loop}^{\vartheta}(\omega_{st}^{\varphi}) \in \Sigma_{loop}(\omega_{st}^{\varphi})$ is translated to a loop in SCL models $\sigma_{loop}^{\vartheta}(\omega_{scl'}) \in \Sigma_{loop}(\omega_{scl'})$ using the translation function $t_{st \mapsto scl}^{\Sigma_{loop}}(\sigma_{loop}^{\vartheta}(\omega_{st}^{\varphi}))$, which is described by Algorithm 15.*

[14] Both models, ST and SCL, are included in Appendix B.8 and D.7.

Algorithm 15 Translate loop – ST-to-SCL

Input: $\sigma^{\vartheta}_{loop}(\omega^{\varphi}_{st})$

Output: $\sigma^{\vartheta}_{loop}(\omega_{scl'})$

Translation Function $t^{\Sigma_{loop}}_{st\mapsto scl}(\sigma^{\vartheta}_{loop}(\omega^{\varphi}_{st}))$**:**

switch ϑ **do**

case *head* **do**

$$\sigma^{head}_{loop}(\omega_{scl'}) \leftarrow \left\{ \begin{array}{l} \texttt{while(}t^{\tau}_{st\mapsto scl}(\lambda^{b}(\sigma^{head}_{loop}(\omega^{\varphi}_{st})))\texttt{)\{} \\ \quad t^{\Sigma}_{st\mapsto scl}(\Sigma(\sigma^{head}_{loop}(\omega^{\varphi}_{st})) \\ \quad \texttt{pause;} \\ \texttt{\}} \end{array} \right\}$$

end

case *foot* **do**

$$\sigma^{foot}_{loop}(\omega_{scl'}) \leftarrow \left\{ \begin{array}{l} \texttt{do:} \\ \quad t^{\Sigma}_{st\mapsto scl}(\Sigma(\sigma^{foot}_{loop}(\omega^{\varphi}_{st})) \\ \quad \texttt{pause;} \\ \texttt{if(!(}t^{\tau}_{st\mapsto scl}(\lambda^{b}(\sigma^{foot}_{loop}(\omega^{\varphi}_{st})))\texttt{))\{ goto do; \}} \end{array} \right\}$$

end

end

Correctness

To check the correctness of Definition 5.8, the following lemmas are used.

Lemma 5.12. *Let* ω^{φ}_{st} *be translated to* $\omega_{scl'}$ *and* $\varphi \in \{fb, fun, prg\}$. *Then,* $t^{\Sigma_{loop}}_{st\mapsto scl}(\sigma^{head}_{loop}(\omega^{\varphi}_{st}))$ *translates* $\sigma^{head}_{loop}(\omega^{\varphi}_{st})$ *to* $\sigma^{head}_{loop}(\omega_{scl'})$ *as specified in Definition 5.8.* $\sigma^{head}_{loop}(\omega_{scl'})$ *conforms to the syntax rules of* $\sigma^{head}_{loop}(\omega_{scl})$ *and preserves the SOS rules of* $\sigma^{head}_{loop}(\omega^{\varphi}_{st})$.

Proof The validity of Lemma 5.12 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma^{head}_{loop}(\omega_{scl'})$ with the syntax rules of $\sigma^{head}_{loop}(\omega_{scl})$ as specified in Section 3.4. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma^{head}_{loop}(\omega_{scl})$ in Section 3.4 with the SOS transition rules of $\sigma^{head}_{loop}(\omega^{\varphi}_{st})$ in Section 3.2.

Illustrative Example for Lemma 5.12

As an example, the head-controlled loop of the `ST_LOOP_HEAD` model is translated as follows[15]:

```
1    i:=i0;                    // initial loop
2    WHILE i<=i1 DO
```

[15] Both models, ST and SCL, are included in Appendix B.14 and D.12.

```
3           y := i;
4           i := i+x2;
5       END_WHILE;
6       y := x1;
7           ⇒ i = i0;           // resulting loop
8           ⇒ while(i<=i1){
9           ⇒    y = i;
10          ⇒    i = i + x2;
11          ⇒ }
12          ⇒ y = x1;
```

Lemma 5.13. *Let ω_{st}^{φ} be translated to $\omega_{scl'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t_{st \mapsto scl}^{\Sigma_{loop}}(\sigma_{loop}^{foot}(\omega_{st}^{\varphi}))$ translates $\sigma_{loop}^{foot}(\omega_{st}^{\varphi})$ to $\sigma_{loop}^{foot}(\omega_{scl'})$ as specified in Definition 5.8. $\sigma_{loop}^{foot}(\omega_{scl'})$ conforms to the syntax rules of $\sigma_{loop}^{foot}(\omega_{scl})$ and preserves the SOS rules of $\sigma_{loop}^{foot}(\omega_{st}^{\varphi})$.*

Proof The validity of Lemma 5.13 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma_{loop}^{foot}(\omega_{scl'})$ with the syntax rules of $\sigma_{loop}^{foot}(\omega_{scl})$ as specified in Section 3.4. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{loop}^{foot}(\omega_{scl})$ in Section 3.4 with the SOS transition rules of $\sigma_{loop}^{foot}(\omega_{st}^{\varphi})$ in Section 3.2.

Illustrative Example for Lemma 5.13

As an example, the foot-controlled loop of the `ST_LOOP_FOOT` model is translated as follows[16]:

```
1       i:=i0;                  // initial loop
2       REPEAT
3           y := x0;
4           i := i + x2;
5       UNTIL i>i1
6       END_REPEAT;
7       y := x1;
8           ⇒ i = i0;           // resulting loop
9           ⇒ do:
10          ⇒    y = x0;
11          ⇒    i = i + x2;
12          ⇒    pause;
13          ⇒ if(!(i>i1)){
14          ⇒    goto do;
15          ⇒ }
16          ⇒ y = x1;
```

5.2.9. Sequences

This step covers the translation function for translating sequences in ST models $\Sigma_{seq}(\omega_{st}^{\varphi})$ to sequences in SCL models $\Sigma_{seq}(\omega_{scl'})$.

[16] Both models, ST and SCL, are included in Appendix B.13 and D.11.

Definition 5.9 (Translation of sequences – ST-to-SCL). *Let* $\Omega_{st}^{\varphi} = \{\omega_{st}^{\varphi} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible ST model elements. There are the following variants of ST statements considered in this approach:*

- *Assignments:* $\sigma_{ass}^{imm}(\omega_{st}^{\varphi}) \in \Sigma_{ass}^{imm}(\omega_{st}^{\varphi})$, $\sigma_{ass}^{del}(\omega_{st}^{\varphi}) \in \Sigma_{ass}^{del}(\omega_{st}^{\varphi})$
- *Conditions:* $\sigma_{cond}^{head}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{head}(\omega_{st}^{\varphi})$, $\sigma_{cond}^{foot}(\omega_{st}^{\varphi}) \in \Sigma_{cond}^{foot}(\omega_{st}^{\varphi})$
- *Loops:* $\sigma_{loop}^{head}(\omega_{st}^{\varphi}) \in \Sigma_{loop}^{head}(\omega_{st}^{\varphi})$, $\sigma_{loop}^{foot}(\omega_{st}^{\varphi}) \in \Sigma_{loop}^{foot}(\omega_{st}^{\varphi})$

According to the Definitions 5.6, 5.7, and 5.8, these ST statements are translated to the following variants of SCL statements:

- *Pause:* $\sigma_{pause}(\omega_{scl'}) \in \Sigma_{pause}(\omega_{scl'})$
- *Assignments:* $\sigma_{ass}^{imm}(\omega_{scl'}) \in \Sigma_{ass}^{imm}(\omega_{scl'})$, $\sigma_{ass}^{del}(\omega_{scl'}) \in \Sigma_{ass}^{del}(\omega_{scl'})$
- *Conditions:* $\sigma_{cond}^{head}(\omega_{scl'}) \in \Sigma_{cond}^{head}(\omega_{scl'})$, $\sigma_{cond}^{foot}(\omega_{scl'}) \in \Sigma_{cond}^{foot}(\omega_{scl'})$
- *Loops:* $\sigma_{loop}^{head}(\omega_{scl'}) \in \Sigma_{loop}^{head}(\omega_{scl'})$, $\sigma_{loop}^{foot}(\omega_{scl'}) \in \Sigma_{loop}^{foot}(\omega_{scl'})$

A set of the resulting SCL statements represents a sequence, denoted as $\Sigma_{seq}(\omega_{scl'})$. *The translation function* $t_{st\mapsto scl}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{scl'}))$ *inserts* $\sigma_i(\omega_{scl'}) \in \Sigma_{seq}(\omega_{scl'})$ *to* $\omega_{scl'}$, *following the process described by Algorithm 16.*

Algorithm 16 Add sequence – ST-to-SCL

Input: $\Sigma_{seq}(\omega_{scl'})$
Output: $\omega_{scl'}$
Translation Function $t_{fbd\mapsto scl}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{scl'}))$**:**

```
forall σ_i ∈ Σ_seq(ω_scl') do
    ω_scl' ← add σ_i to the position w.r.t. its execution order and dependent
      constructs
end
```

Correctness

To check the correctness of Definition 5.9, the following lemma is used.

Lemma 5.14. *Let* ω_{st}^{φ} *be translated to* $\omega_{scl'}$ *and* $\varphi \in \{fb, fun, prg\}$, $\Sigma_{seq}(\omega_{scl'}) \neq \varnothing$, *and* $\Sigma_{seq}(\omega_{scl'})$ *be syntactically correct. Then,* $t_{fbd\mapsto scl}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{scl'}))$ *inserts each translated statement* $\sigma_i \in \Sigma_{seq}(\omega_{scl'})$ *to* $\omega_{scl'}$ *as specified in Definition 5.9. The resulting SCL model* $\omega_{scl'}$ *conforms to the syntax rules of* ω_{scl} *and preserves the SOS rules of* ω_{st}^{φ} *(in particular with regard to the execution order of the statements).*

Proof The validity of Lemma 5.14 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{scl'})$ with the syntax rules of $\Sigma_{seq}(\omega_{scl})$ as specified in Section 3.4 using induction on the number of statements to be added $\Sigma_{seq}(\omega_{scl'})$:

1. **Base Case:** When $\Sigma_{seq}(\omega_{scl'}) = \varnothing$, there are no statements to be added, which conforms to the syntax rules of $\delta_{\omega}(\omega_{scl})$.
2. **Induction Hypothesis:** The lemma holds for any set of statements to be added $\Sigma_{seq}(\omega_{scl'})$.
3. **Inductive Step:** Adding a statement results in a statement to be added that conforms to the syntax rules $\Sigma_{seq}(\omega_{scl})$, because the syntactic correctness of this statement to be added has been proved in the corresponding section.

Second, the SOS transition rules are respected by the individual statements themselves. The order is respected by the order within the resulting *Quartz* model.

Illustrative Example for Lemma 5.14

As an example, below are the inserted statements in the resulting SCL model of the `ST_TON` model[17]:

```
1  ...
2  if(IN != LASTIN){                          ⇐ σ1 ∈ Σ^ite_cond(ω_scl')
3      LASTIN = IN;                           ⇐ σ2 ∈ Σ^imm_ass(ω_scl')
4      if(IN){                                ⇐ σ3 ∈ Σ^ite_cond(ω_scl')
5        TSTART = CLK;                        ⇐ σ4 ∈ Σ^imm_ass(ω_scl')
6      }else{
7        TSTART = 0;                          ⇐ σ5 ∈ Σ^imm_ass(ω_scl')
8      }
9      Q1_Temp1 = false;                      ⇐ σ6 ∈ Σ^imm_ass(ω_scl')
10     Q1 = Q1_Temp1;                         ⇐ σ7 ∈ Σ^imm_ass(ω_scl')
11     ET = 0;                                ⇐ σ8 ∈ Σ^imm_ass(ω_scl')
12 }else{
13   if(IN & (!(Q1_Temp1))){                  ⇐ σ9 ∈ Σ^ite_cond(ω_scl')
14     ETTIME = (CLK - TSTART);               ⇐ σ10 ∈ Σ^imm_ass(ω_scl')
15     if(ETTIME < PT){                       ⇐ σ11 ∈ Σ^ite_cond(ω_scl')
16       ET = ETTIME;                         ⇐ σ12 ∈ Σ^imm_ass(ω_scl')
17     }else{
18       Q1_Temp1 = true;                     ⇐ σ13 ∈ Σ^imm_ass(ω_scl')
19       Q1 = Q1_Temp1;                       ⇐ σ14 ∈ Σ^imm_ass(ω_scl')
20       ET = PT;                             ⇐ σ15 ∈ Σ^imm_ass(ω_scl')
21     }
22   }
23 }
24 ...
```

5.3. Experimental Results

The applicability of the introduced translation functions is evaluated with the ST models listed in Table 5.1. The examples are listed with the SLOC metric [BM14] of the ST model and the experimental results. For evaluation purposes, the listed ST models and the expected SCL models are manually

[17] Both models, ST and SCL, are included in Appendix B.22 and D.19.

implemented to verify the applicability of the isolated translation functions. To ensure the correctness of both models, ST and SCL, they are compiled with the built-in compilers of *CODESYS* and *KIELER*. The translation functions have been implemented as a prototype in *PLCreX*, resulting in the overall test strategy shown in Figure 5.5. The correctness of the resulting SCL models is verified in two ways: (1) through manual reviews, differences between the expected SCL models and the automatically generated models are identified, and (2) using the built-in compilers of *KIELER*, the syntactic correctness of the automatically generated SCL models is ensured. Both tests passed for all ex-

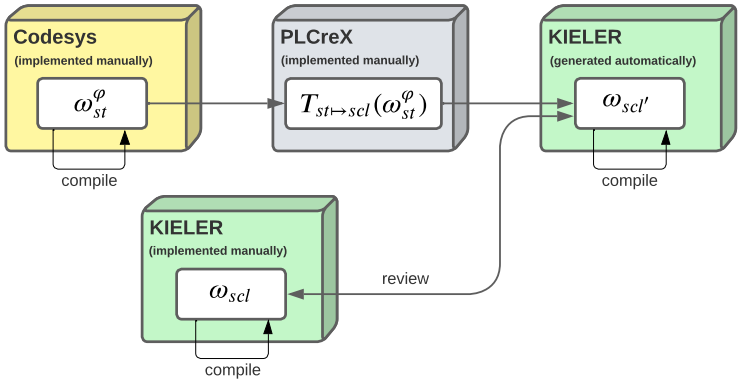


Figure 5.5.: *Test strategy to evaluate the ST-to-SCL transformation*

amples (ignoring minor formatting differences between manually implemented and automatically generated SCL models), with the following warnings:

- **Supported operators**: According to Lemma 5.8, only a subset of the considered ST expressions can be translated into corresponding SCL expressions with the restriction to internal operators. Therefore, the example *arithmetic operators* throws a warning for affected expressions. Affected expressions are skipped during translation.
- **Supported data types**: According to Lemma 5.7, only a subset of the considered ST data types can be translated into corresponding SCL data types with the restriction to internal data types. Therefore, the example *data types and fields* throws a warning for affected declarations. Affected declarations and expressions are skipped during translation.

Furthermore, some examples are not testable:

- **Model imports**: Based on experimental results in the latest version of *KIELER*, SCL models are not intended to import external SCL models, which is why the examples *analog value processing 2*, *debounce*, *invoke function 1*, *simple program*, and *track correction* are not testable and are not listed in Appendix D. ST models that import other models are not translated to SCL models by default.

Based on the experimental results in Table 5.1, it can be concluded that the introduced translation functions are applicable and lead to correct SCL models

Table 5.1.: *Set of ST models and test results to evaluate the applicability of the introduced ST-to-SCL transformation*

Model	SLOC	Source	ω_{st}^{φ}	$\omega_{scl'}$	Result
2-of-3 logic function	11	[Sch19]	B.1	D.1	passed
Alarm function	8	[Sch19]	B.2	D.2	passed
Analog value processing 1	8	[Sch19]	B.3	D.3	passed
Analog value processing 2	7	[Sch19]	B.4	-	not testable
Arithmetic operators	20	self	B.5	D.4	passed with warnings
Boolean operators	14	self	B.6	D.5	passed
Compensation system	28	[Sch19]	B.7	D.6	passed
Condition statements	15	self	B.8	D.7	passed
Data types and fields	28	self	B.9	D.8	passed with warnings
Debounce	20	[GDV14]	B.10	-	not testable
Delayed assignments	7	self	B.11	D.9	passed
Equality, inequality	9	self	B.12	D.10	passed
Foot-controlled loop	20	self	B.13	D.11	passed
Head-controlled loop	18	self	B.14	D.12	passed
Immediate assignments 1	13	self	B.15	D.13	passed
Immediate assignments 2	19	self	B.16	D.14	passed
Immediate assignments 3	10	self	B.17	D.15	passed
Invoke function 1	7	self	B.18	-	not testable
Left detection	7	[Sch19]	B.19	D.16	passed
Numeric relations	11	self	B.20	D.17	passed
Off delay timer	39	[GDV14]	B.21	D.18	passed
On delay timer	39	[GDV14]	B.22	D.19	passed
RS-Flip-Flop	14	[GDV14]	B.23	D.20	passed
Right detection	7	[Sch19]	B.24	D.21	passed
SR-Flip-Flop	14	[GDV14]	B.25	D.22	passed
Simple calculation	16	[GDV14]	B.26	D.23	passed
Simple program	19	[GDV14]	B.27	-	not testable
Tank control	19	[Sch19]	B.28	D.24	passed
Track correction	23	[Sch19]	B.29	-	not testable
Two-Point controller	21	[Sch19]	B.30	D.25	passed

that can be reused in model-based design, if a few conditions are considered. These are summarized in the following section.

5.4. Summary

This chapter introduced the transformation of ST models to SCL models. For this purpose, individual translation functions were defined that take into

account the sequential execution order of ST statements. The applicability of these translation functions was demonstrated using a set of ST examples. Based on the presented lemmas and experimental results, the following theorem encapsulates the entire transformation:

Theorem 5.1 (ST-to-SCL Translation). *Let $\omega_{st}^{\varphi} \in \Omega_{st}^{\varphi}$ be an ST model of the variant $\varphi \in \{fb, fun, prg\}$ and let $T_{st \mapsto scl}(\omega_{st}^{\varphi})$ be the model transformation of ω_{st}^{φ} to $\omega_{scl'}$ using the translation functions defined in this chapter. Then, the resulting SCL model $\omega_{scl'}$:*

1. *Conforms to the syntax rules of ω_{scl}*
2. *Preserves the semantics of ω_{st}^{φ}*
3. *Contains constructs corresponding to the constructs of ω_{st}^{φ} and preserves the intended functionality of ω_{st}^{φ} under the following conditions:*
   - $\varphi \in \{fb, fun, prg\}$
   - $\Delta_{idcl}(\omega_{st}^{\varphi}) = \Delta_{in}(\omega_{st}^{\varphi}) \cup \Delta_{out}(\omega_{st}^{\varphi}) \cup \Delta_{inout}(\omega_{st}^{\varphi})$
   - $\Delta_{vdcl}(\omega_{st}^{\varphi}) = \Delta_{local}(\omega_{st}^{\varphi})$
   - $\forall \alpha^{[+]}(\omega_{st}^{\varphi}) : \alpha^{[+]} \in \{\alpha_{bv}^{bool}, \alpha_{i}^{int}, \alpha_{i}^{dint}, \alpha_{i}^{uint}, \alpha_{dur}, \alpha^{+}\}$, *where α_{dur} can be treated as a bounded integer and is specified in milliseconds*
   - $\forall \tau(\omega_{st}^{\varphi}) : \tau \in \{\tau_{misc}^{cst}, \tau_{misc}^{id}, \tau_{misc}^{\pi,\eta,\lambda}, \tau_{misc}^{br}, \tau_{misc}^{true}, \tau_{misc}^{false}, \tau_{misc}^{arr}, \tau_{misc}^{inv}, \tau_{comp}^{eq}, \tau_{comp}^{ne}, \tau_{comp}^{gt}, \tau_{comp}^{ge}, \tau_{comp}^{lt}, \tau_{comp}^{le}, \tau_{arith}^{mul}, \tau_{arith}^{div}, \tau_{arith}^{add}, \tau_{arith}^{sub}, \tau_{arith}^{expt}, \tau_{arith}^{mod}, \tau_{arith}^{um}, \tau_{cond}^{sel}\}$
   - $\forall \sigma(\omega_{st}^{\vartheta}) : \sigma \in \{\sigma_{ass}^{\vartheta}, \sigma_{cond}^{\vartheta}, \sigma_{loop}^{\vartheta}\}$

Proof The validity of Theorem 5.1 is proved as follows:

1. **Syntax Conformance**: Lemma 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13, and 5.14 demonstrate that each translated construct conforms to the syntax rules of ω_{scl} as specified in Section 3.4.
2. **Semantic Preservation**: The following lemmas address the preservation of semantics for their respective constructs.
   - Model declaration: Lemma 5.1
   - Interfaces: Lemma 5.2
   - Variables: Lemma 5.6
   - Data types and fields: Lemma 5.7
   - Expressions: Lemma 5.8

- Assignments: Lemma 5.9 and 5.10
- Conditions: Lemma 5.11
- Loops: Lemmas 5.12 and 5.13
- Sequences: Lemma 5.14

3. **Construct Correspondence**: Given the conditions of the theorem, the provided definitions, proofs, and experimental results, it can be concluded that the translation functions produce corresponding constructs in ω_{scl} for the considered constructs in ω_{st}^{φ}, preserving the original functionality.

Overall, this results in the following solutions to the challenges summarized in Section 5.1.

1. **Cyclic execution of SCL models (with and without memory)**: The execution of the SCL model with and without memory depends on the invocation by an external model, i.e., an external model is responsible for the cyclic execution of the resulting SCL model. Unlike variables of a resulting SCL model based on an ST model with memory, variables of a resulting SCL model based on an ST model without memory are set to their default values (if necessary) at the beginning of each iteration.
2. **Event-driven execution of SCL models**: An iteration of the SCL model is triggered via the input variable `EI` and returned via the output variable `EO` after a finite number of *macro steps* ($n \geq 0$), thus realizing an external event-driven execution control.
3. **Dynamic system time**: The global clock synchronization is realized by an additional input variable `CLK`, i.e., the clock is controlled externally and processed within the SCL model with read access.
4. **Translation of ST language constructs**: The solution follows from Theorem 5.1.

Chapter 6

Model Transformation of FBDs to *Quartz* Models

Contents

The third approach to reusing existing POUs in model-based design is the transformation of FBDs into *Quartz* models, where the goal is to create a robust set of translation functions that ensure semantic preservation during the transition. In addition to the approaches presented in [WS20; WS21; WS24b], it considers a number of additional aspects as mentioned in Chapter 4 in the context of the ST-to-Quartz transformation. The correctness of the translation functions proposed in this chapter is proved by theoretical reasoning, which includes a detailed analysis of the resulting syntax and semantics compared to the syntax rules and semantics specified in Chapter 3. In addition, the theoretical results are evaluated with real-world and self-defined FBDs.

This chapter is structured as follows: Section 6.1 introduces the high-level design flow and translation strategy. Section 6.2 defines the translation func-

tions and theoretical analysis. Section 6.3 presents an evaluation of the theoretical results, and Section 6.4 summarizes the transformation.

6.1. High-Level Design Flow – FBD-to-Quartz

The high-level design flow for transforming an FBD $\omega_{fbd}^{\varphi} \in \Omega_{fbd}^{\varphi}$ of the variant *function block* ($\varphi = fb$), *function* ($\varphi = fun$), or *program* ($\varphi = prg$) to a *Quartz* model $\omega_{qrz} \in \Omega_{qrz}$ is shown in Figure 6.1.

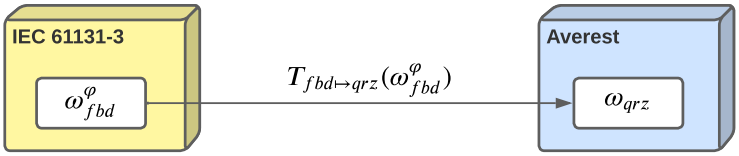


Figure 6.1.: *High-level design flow of the FBD-to-Quartz transformation*

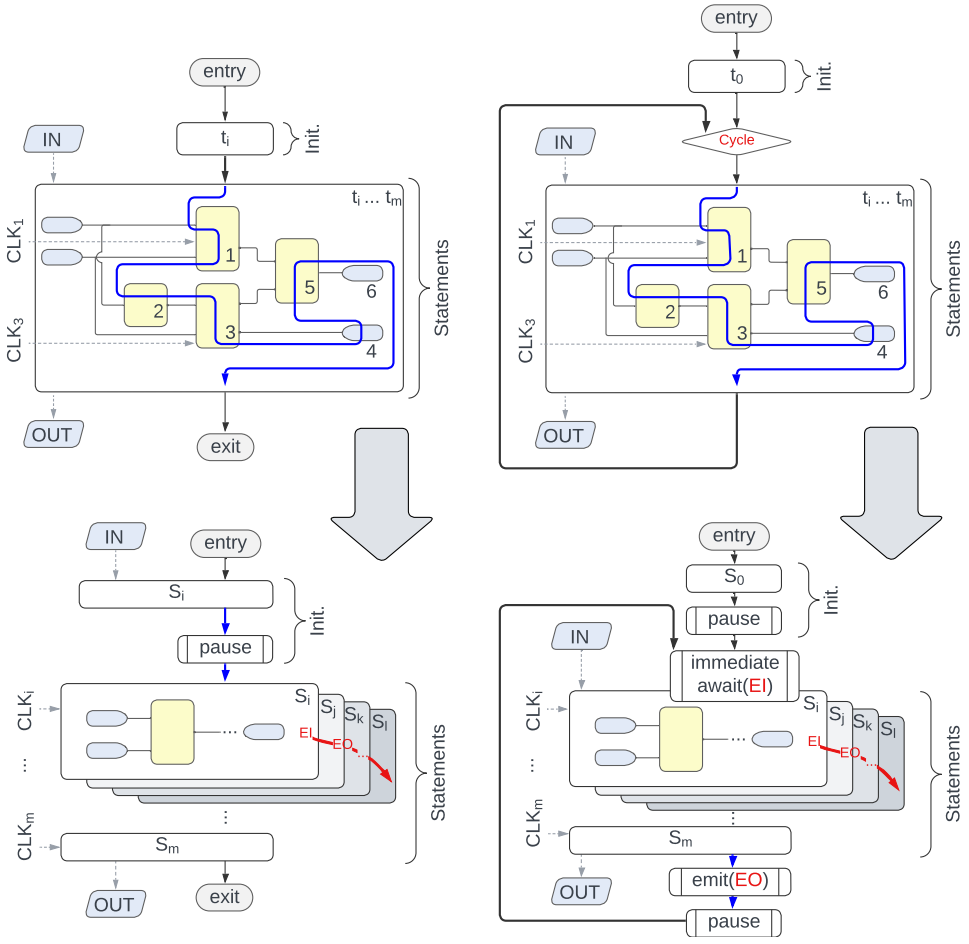
The FBD-to-Quartz transformation $T_{fbd \mapsto qrz}(\omega_{fbd}^{\varphi})$ includes the following transformation steps:

1. $t_{fbd \mapsto qrz}^{\delta_\omega}(\delta_\omega(\omega_{fbd}^{\varphi}))$: Model declaration (see Section 6.2.1)
2. $t_{fbd \mapsto qrz}^{\Delta_{idcl}}(\omega_{fbd}^{\varphi})$: Interfaces (see Section 6.2.2)
3. $t_{fbd \mapsto qrz}^{\Delta_{vdcl}}(\omega_{fbd}^{\varphi})$: Variables (see Section 6.2.3)
4. $t_{fbd \mapsto qrz}^{\alpha}(\alpha^{[+]}(\omega_{fbd}^{\varphi}))$: Data types and fields (see Section 6.2.4)
5. $t_{fbd \mapsto qrz}^{\Delta_{imports}}(\omega_{fbd}^{\varphi})$: POU imports (see Section 6.2.5)
6. $t_{fbd \mapsto qrz}^{\tau}(\tau(\omega_{fbd}^{\varphi}))$: Expressions (see Section 6.2.6)
7. $t_{fbd \mapsto qrz}^{\Sigma_{inv}}(\sigma_{inv}^{\vartheta}(\omega_{fbd}^{\varphi}))$: POU invocations (see Section 6.2.7)
8. $t_{fbd \mapsto qrz}^{\Sigma_{ass}}(\sigma_{ass}^{\vartheta}(\omega_{fbd}^{\varphi}))$: Assignments (see Section 6.2.8)
9. $t_{fbd \mapsto qrz}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{qrz'}))$: Sequences (see Section 6.2.9)

Translation Strategy:

This chapter introduces two translation strategies that are basically the same as the translation strategies introduced in Section 4.1. The difference is the translation of statements that are given as graphical FBDs and not as textual ST statements. Taking into account an explicit execution order of the blocks and assignments, this results in the two translation strategies illustrated in Figure 6.2, where graphical blocks are used to demonstrate the relationship to the underlying FBD. Figure 6.2a shows the high-level runtime behavior of the initial FBD (top) and the resulting *Quartz* model (bottom) that follows the first translation strategy, where the set of input variables `IN` is read when the model is invoked at time t_i (FBD) or in *macro step* S_i (SCChart), respectively, and where the set of output variables `OUT` is returned at time t_m (FBD)

or in the final *macro step* S_m (SCChart). In contrast, Figure 6.2b shows the high-level runtime behavior of the initial FBD (top) and the resulting *Quartz* model (bottom) that follows the second translation strategy, where the model is initialized at time t_0 (FBD) or in the first *macro step* S_0 (SCChart), respectively, and then waits until an iteration is triggered (SCChart via input variable `EI`). At time t_m (FBD) or in *macro step* S_m (SCChart), `OUT` and `EO` are returned to the invoking model for external processing. The final *macro step* in the resulting SCChart is used to switch to the next iteration (triggered in the next PLC cycle).



(a) *Model of a POU without memory* **(b)** *Model of a POU with memory*

Figure 6.2.: *FBD-to-Quartz translation strategies: high-level runtime behavior of the initial FBD (top) and resulting* Quartz *model (bottom)*

Challenges:

Consequently, this leads to the following challenges for translating FBDs to *Quartz* models, which are almost the same as for translating ST models to *Quartz* models:

1. Cyclic execution of *Quartz* models (with and without memory)
2. Event-driven execution of synchronous parallel threads
3. Sequential execution of synchronous parallel threads
4. Dynamic system time
5. Translation of FBD language constructs

6.2. From FBDs to *Quartz* Models

This section defines the individual translation functions for translating an FBD $\omega^{\varphi}_{fbd} \in \Omega^{\varphi}_{fbd}$ to a *Quartz* model $\omega_{qrz'} \in \Omega_{qrz}$ and analyzes the theoretical correctness.

6.2.1. Model Declaration

The translation strategy for an ST model declaration to a *Quartz* model declaration introduced in Section 4.2.1 can be applied to an FBD $\delta_{\omega}(\omega^{\varphi}_{fbd})$, because the declaration is equivalent [GDV14; PLC09]. As a result, Definition 4.1 and Lemma 4.1 are also valid for FBDs (if all occurrences of ω^{φ}_{st} are replaced by ω^{φ}_{fbd}), resulting in $t^{\delta_{\omega}}_{fbd\mapsto qrz}(\delta_{\omega}(\omega^{\varphi}_{fbd})) \coloneqq t^{\delta_{\omega}}_{st\mapsto qrz}(\delta_{\omega}(\omega^{\varphi}_{fbd}))$.

6.2.2. Interfaces

The translation strategy for ST model interfaces introduced in Section 4.2.2 can be applied to FBD interfaces $\Delta_{idcl}(\omega^{\varphi}_{fbd}) = \Delta_{in}(\omega^{\varphi}_{fbd}) \cup \Delta_{out}(\omega^{\varphi}_{fbd}) \cup \Delta_{inout}(\omega^{\varphi}_{fbd})$, because the declaration is equivalent [GDV14; PLC09]. As a result, Definition 4.2 and Lemma 4.2, 4.3, 4.4, and 4.5 are also valid for FBDs (if all occurrences of ω^{φ}_{st} are replaced by ω^{φ}_{fbd}), resulting in $t^{\Delta_{idcl}}_{fbd\mapsto qrz}(\omega^{\varphi}_{fbd}) \coloneqq t^{\Delta_{idcl}}_{st\mapsto qrz}(\omega^{\varphi}_{fbd})$.

6.2.3. Variables

The translation strategy for local ST model variables introduced in Section 4.2.3 can be applied to FBD variables $\Delta_{vdcl}(\omega^{\varphi}_{fbd}) = \Delta_{local}(\omega^{\varphi}_{fbd})$, because the declaration is equivalent [GDV14; PLC09]. As a result, Definition 4.3 and Lemma 4.6, 4.7, 4.8, and 4.9 are also valid for FBDs (if all occurrences of ω^{φ}_{st} are replaced by ω^{φ}_{fbd}), resulting in $t^{\Delta_{vdcl}}_{fbd\mapsto qrz}(\omega^{\varphi}_{fbd}) \coloneqq t^{\Delta_{vdcl}}_{st\mapsto qrz}(\omega^{\varphi}_{fbd})$.

6.2.4. Data Types and Fields

The translation strategy for ST data types and fields introduced in Section 4.2.4 can be applied to FBD data types and fields $\mathcal{A}^{[+]}(\omega^{\varphi}_{fbd})$, because the data types and fields are equivalent [GDV14]. As a result, Definition 4.4 and Lemma 4.10 are also valid for FBDs (if all occurrences of ω^{φ}_{st} are replaced by ω^{φ}_{fbd}), resulting in $t^{\alpha}_{fbd\mapsto qrz}(\alpha^{[+]}(\omega^{\varphi}_{fbd})) \coloneqq t^{\alpha}_{st\mapsto qrz}(\alpha^{[+]}(\omega^{\varphi}_{fbd}))$.

6.2.5. POU Imports

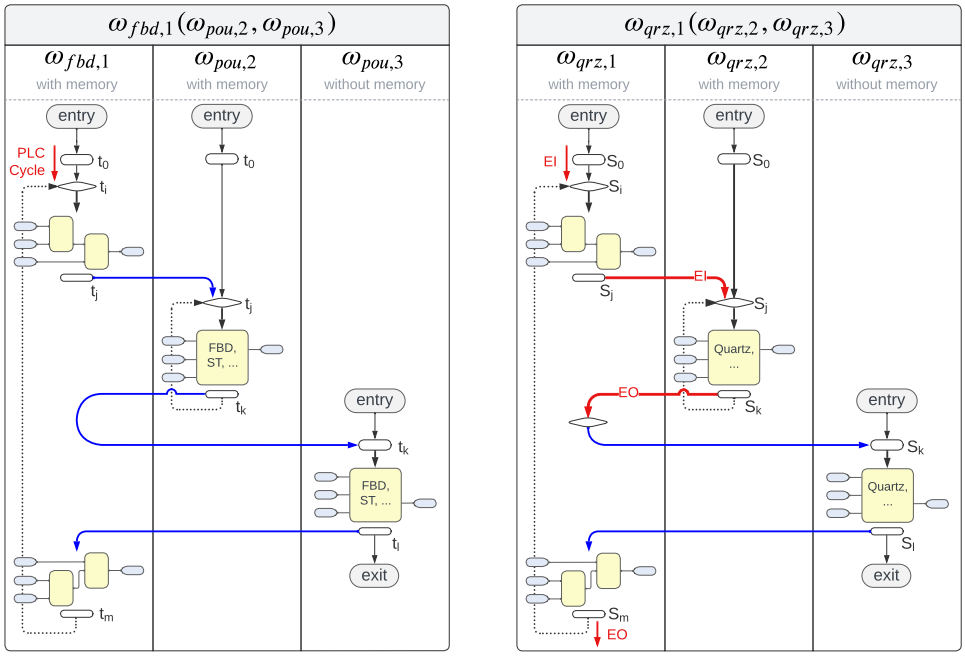
The translation strategy for instantiated and invoked POUs in ST models introduced in Section 4.2.5 can be applied to FBDs, because the instantiated and invoked POUs in FBDs are handled in the same way as in ST models, such as instances of POUs with memory represented as local variables and POUs without memory invoked inline without a specific identifier [GDV14]. As a result, Definition 4.5 and Lemma 4.11 are also valid for FBDs (if all occurrences of ω_{st}^{φ} are replaced by ω_{fbd}^{φ}), resulting in $t_{fbd\mapsto qrz}^{\Delta_{imports}}(\omega_{fbd}^{\varphi}) \coloneqq t_{st\mapsto qrz}^{\Delta_{imports}}(\omega_{fbd}^{\varphi})$.

6.2.6. Expressions

The translation strategy of ST model expressions introduced in Section 4.2.6 can be applied to FBD expressions $\mathcal{T}(\omega_{fbd}^{\varphi})$, because the translations considered by the algorithm are linked to the individual, specified syntax rules and semantics of FBDs. Thus, Definition 4.6 and Lemma 4.12 are also valid for FBDs (if all occurrences of ω_{st}^{φ} are replaced by ω_{fbd}^{φ}), resulting in $t_{fbd\mapsto qrz}^{\tau}(\tau(\omega_{fbd}^{\varphi})) \coloneqq t_{st\mapsto qrz}^{\tau}(\tau(\omega_{fbd}^{\varphi}))$.

6.2.7. POU Invocations

This step covers the translation function for translating POU invocations in FBDs $\Sigma_{inv}^{\vartheta}(\omega_{fbd}^{\varphi})$ to *Quartz* model invocations in *Quartz* models $\Sigma_{inv}^{\vartheta}(\omega_{qrz'})$. The high-level runtime behavior of FBD (with memory) translated to a *Quartz* model is similar to the high-level runtime behavior introduced in Section 4.2.7 with the difference that the statements and their execution order are not present as textual statements and position, but as graphical FBDs and execution order identifiers. Therefore, the adapted high-level runtime behavior is illustrated in Figure 6.3. Figure 6.3a shows the runtime behavior of an example FBD with memory that is triggered in each PLC cycle (red) and invokes a model with memory and a model without memory depending on their execution order (blue). In contrast, Figure 6.3b shows the runtime behavior of the resulting *Quartz* model, whose models with memory are triggered by additional event-driven variables (red) and the model without memory depending on its execution order (blue) without an additional event-driven variable. As a consequence, the translation strategy for invocations in ST models introduced in Section 4.2.7 can be applied to FBDs, taking into account the graphical representation of the expressions and invocations. More specifically, Definition 4.7 can be applied to FBDs (if all occurrences of ω_{st}^{φ} are replaced by ω_{fbd}^{φ}) with minor changes: Unlike the precondition in Definition 4.7, POU invocations are not represented as *complete formal function call* [GDV14], but all ports are visible in the graphical FBD, i.e., the information is the same. However, the difference is that an argument i of an invoked model is not given as a string. For this reason, i must be identified before it is placed. This is described by Algorithm 17, which describes a backward translation strategy introduced in [YKL13]. As a consequence, the following replacement in Algorithm 5 must be considered for



(a) *Initial FBD* **(b)** *Resulting* Quartz *model*

Figure 6.3.: *High-level runtime behavior of a model with memory that invokes two models (Approach: FBD-to-Quartz)*

FBDs: $t^{\tau}_{st\mapsto qrz}(i) \mapsto t^{\tau}_{st\mapsto qrz}(\texttt{get\_expr}(i))$. Then, Lemma 4.13 and 4.14 are also valid for FBDs, resulting in $t^{\Sigma_{inv}}_{fbd\mapsto qrz}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})) :\equiv t^{\Sigma_{inv}}_{st\mapsto qrz}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))$.

Algorithm 17 Get expression following backward translation strategy

Input: lhs
Output: $\tau_i(\omega^{\varphi}_{fbd'})$
Function *get_expr*(lhs)**:**
  find $rhs \in \mathcal{E}_{iV} \cup \mathcal{E}_{iOV} \cup \mathcal{E}_{b^{inst}} \cup \mathcal{E}_{b^{fun}} \cup \mathcal{E}_{b^{fun'}}$, where $a_{rLI}(lhs) = a_{lI}(rhs)$;
  if $rhs \notin \mathcal{E}_{b^{fun}}$ **then**
    return $\tau_i(\omega^{\varphi}_{fbd'}) = eval(a_{fP}(rhs), a_{lI}(rhs), e_e(rhs))$;
  else
    $rhs \leftarrow a_{tN}(rhs)$ with parameter list $\mathcal{E}_{iVs}(rhs) \cup \mathcal{E}_{iOVs}(rhs)$;
    forall $lhs^i \in \mathcal{E}_{iVs}(rhs) \cup \mathcal{E}_{iOVs}(rhs)$ **do**
      $lhs^i \leftarrow$ assign *get_expr*(lhs^i);
      update rhs;
    end
    return $\tau_i(\omega^{\varphi}_{fbd'}) = rhs$;
  end

Correctness

To check the correctness of the function `get_expr(`lhs`)`, the following lemma is used.

Lemma 6.1. *Let* ω_{fbd}^{φ} *be translated to* $\omega_{qrz'}$, $\varphi \in \{fb, fun, prg\}$, *and* ω_{fbd}^{φ} *does not contain explicit loops. Then, the function* `get_expr(lhs)` *described by Algorithm 17 identifies and returns the right-hand side expression* $\tau_i(\omega_{fbd'})$ *for* $lhs = element(x)$ *with* $x \in \{\delta_{out}(\omega_{fbd}^{\varphi}), \delta_{inout}(\omega_{fbd}^{\varphi}), \delta_{local}(\omega_{fbd}^{\varphi}), \delta_{in}(\omega_{pou,i}^{\varphi})$ *with* $\omega_{pou,i}^{\varphi} \neq \omega_{fbd}^{\varphi}\}$, *where* $\tau_i(\omega_{fbd'}^{\varphi})$ *conforms to the syntax rules of* $\tau_i(\omega_{fbd}^{\varphi})$ *and preserves the semantics of* $\tau_i(\omega_{fbd}^{\varphi})$ *regarding the I/O behavior.*

Proof The validity of Lemma 6.1 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\tau(\omega_{qrz'})$ with the syntax rules of $\tau(\omega_{qrz})$ as specified in Section 3.3 using inductive reasoning.

1. **Base Case:** According to the specifications for a valid *PLCopen* `xml` file [PLC09], there is a unique link between $a_{rLI}(lhs)$ and $a_{lI}(rhs)$ of FBD elements. Thus, a call of `get_expr(`lhs`)` with $lhs = element(x)$, $x \in \{\delta_{out}(\omega_{fbd}^{\varphi}), \delta_{inout}(\omega_{fbd}^{\varphi}), \delta_{local}(\omega_{fbd}^{\varphi}), \delta_{in}(\omega_{plc,i}^{\varphi})$ with $\omega_{plc,i}^{\varphi} \neq \omega_{fbd}^{\varphi}\}$ always leads to a corresponding rhs. If $rhs \notin \mathcal{E}_{bfun}$, $eval(a_{fP}(rhs), a_{lI}(rhs), e_e(rhs))$ is returned, which is a variable name or a constant. This terminates the algorithm.
2. **Inductive Hypothesis:** For any right-hand side parameter list expression $lhs^i \in \mathcal{E}_{iVs}(rhs) \cup \mathcal{E}_{iOVs}(rhs)$, the function `get_expr(`$lhs^i$`)` syntactically correctly derives a rhs^i, so that the algorithm terminates after a finite number of steps.
3. **Inductive Step:** Considering the recursive case where $rhs \in \mathcal{E}_{bfun}$. The algorithm replaces rhs through the block name $a_{tN}(rhs)$ with parameter list $\mathcal{E}_{iVs}(rhs) \cup \mathcal{E}_{iOVs}(rhs)$ and recursively derives expressions for parameter list elements $lhs^i \in \mathcal{E}_{iVs}(rhs) \cup \mathcal{E}_{iOVs}(rhs)$, ensuring that each recursive call maintains the appropriate $a_{rLI}(lhs^i) = a_{lI}(rhs^i)$ relationship according to the inductive hypothesis. The correctness of these recursive steps ensures that the final rhs expression correctly matches lhs as required.

   Thus, after executing the function, there are the following possible post conditions:

   a) If $rhs \notin \mathcal{E}_{bfun}$, $eval(a_{fP}(rhs), a_{lI}(rhs), e_e(rhs))$ is returned, which is a variable name or a constant. It also terminates the algorithm.

   b) If $rhs \in \mathcal{E}_{bfun}$, the rhs is correctly updated and replaced based on recursive outcomes and transformations, maintaining the relationship $a_{rLI}(lhs) = a_{lI}(rhs)$, and returns final rhs.

Since ω_{fbd}^{φ} does not contain explicit loops by definition, the algorithm terminates after a finite number of steps. Second, the semantic correctness is given because this algorithm implements the backward translation strategy, whose general correctness with respect to I/O behavior has been demonstrated in [YKL13].

Illustrative Example for Lemma 6.1

As an illustrative example of how `get_expr(`*lhs*`)` is processed, below is a snippet of the trace when `get_expr(`*lhs*`)` is called for the `FBD_AIR_COND_CTRL`[1] model, parameter `RS0.RESET1`, i.e., *lhs* = `RS0.RESET1`. The order in which `get_expr(RS0.RESET1)` is processed is visualized in Figure 6.4:

```
1  get_expr(RS0.RESET1):                              // 1. call
2  get_expr(OR(<IN1>, <IN2>)):                        // replace
3    get_expr(<IN1>):                                 // 2. call
4    get_expr(OR(<IN1>, <IN2>, <IN3>, <IN4>):         // replace
5      get_expr(<IN1>):                               // 3. call
6      get_expr(NOT(<IN1>)):                          // replace
7        get_expr(<IN1>):                             // 4. call
8          ⇒ IN5                                      // return
9          ⇒ NOT(IN5)                                 // update
10     ...
11     get_expr(<IN4>):                               // 9. call
12     get_expr(NOT(<IN1>)):                          // replace
13       get_expr(<IN1>):                             // 10. call
14         ⇒ IN8                                      // return
15         ⇒ NOT(IN8)                                 // update
16     ⇒ OR(NOT(IN5),...,NOT(IN8))                    // update
17   get_expr(<IN2>):                                 // 11. call
18   get_expr(NOT(<IN1>)):                            // replace
19     get_expr(<IN1>):                               // 12. call
20       ⇒ IN9                                        // return
21       ⇒ NOT(IN9)                                   // update
22   ...
23   ⇒ OR(NOT(IN5),...,NOT(IN8)),NOT(IN9))            // return
```

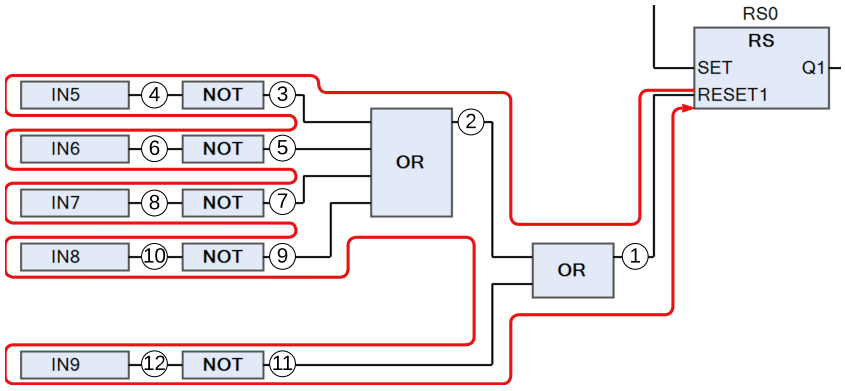


Figure 6.4.: *Visualization of the processing sequence:* `get_expr(RS0.RESET1)`

[1] Both models, FBD and *Quartz*, are included in Appendix E.2 and F.2.

6.2.8. Assignments

This step covers the translation function for translating assignments in FBDs $\Sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd})$ to assignments in *Quartz* models $\Sigma^{\vartheta}_{ass}(\omega_{qrz'})$. The translation of assignments in FBDs is quite similar to the translation of assignments in ST models [GDV14], introduced in Section 4.2.8. For this reason, Definition 4.8 can be applied to FBDs (if all occurrences of ω^{φ}_{st} are replaced by ω^{φ}_{fbd}) with minor changes. The difference is that in FBDs the right-hand side *rhs* must first be extracted by the graphical model (or by elements of the textual *PLCopen* `xml` format, respectively) before it can be processed. As a consequence, the following replacement in Algorithm 6 must be considered for FBDs: $t^{\tau}_{st\mapsto qrz}(lhs(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}))) \mapsto t^{\tau}_{st\mapsto qrz}(\texttt{get\_expr}(lhs(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}))))$. Then, Lemma 4.15 and 4.16 are also valid for FBDs, resulting in $t^{\Sigma_{ass}}_{fbd\mapsto qrz}(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd})) \coloneqq t^{\Sigma_{ass}}_{st\mapsto qrz}(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}))$.

6.2.9. Sequences

The translation strategy for sequences in ST models introduced in Section 4.2.11 can be applied to sequences in FBDs $\Sigma_{seq}(\omega^{\varphi}_{fbd})$, where only the following statements are applicable to FBDs:

- Pause: $\sigma_{pause}(\omega_{qrz'}) \in \Sigma_{pause}(\omega_{qrz'})$
- Model invocations: $\sigma_{inv}(\omega_{qrz'}) \in \Sigma_{inv}(\omega_{qrz'})$
- Assignments: $\sigma^{imm}_{ass}(\omega_{qrz'}) \in \Sigma^{imm}_{ass}(\omega_{qrz'})$, $\sigma^{del}_{ass}(\omega_{qrz'}) \in \Sigma^{del}_{ass}(\omega_{qrz'})$

For this reason, Definition 4.11 (with limited statements) and Lemma 4.20 are also valid for FBDs (if all occurrences of ω^{φ}_{st} are replaced by ω^{φ}_{fbd}, resulting in $t^{\Sigma_{seq}}_{fbd\mapsto qrz}(\omega^{\varphi}_{fbd}) \coloneqq t^{\Sigma_{seq}}_{st\mapsto qrz}(\omega^{\varphi}_{fbd})$.

Due to the equivalence to ST models, additional examples for FBD model declarations, interfaces, variables, data types and fields, POU imports, expressions, POU invocations, and sequences are not presented at this point.

6.3. Experimental Results

The applicability of the introduced translation functions is evaluated with the FBDs listed in Table 6.1. For evaluation purposes, the listed FBDs and the expected *Quartz* models are manually implemented to verify the applicability of the isolated translation functions. To ensure the correctness of both models, FBDs and *Quartz*, they are compiled with the built-in compilers of *CODESYS* and *Averest*. The translation functions have been implemented as a prototype in *PLCreX*, assuming the FBDs are available in *PLCopen* `xml` format, resulting in the overall test strategy shown in Figure 6.5. The correctness of the resulting *Quartz* models is verified in two ways: (1) through manual reviews, differences between the expected *Quartz* models and the automatically generated models are identified, and (2) using the built-in compilers of *Averest*, the syntactic correctness of the automatically generated *Quartz* models is ensured.

Both tests passed for all examples (ignoring minor formatting differences be-

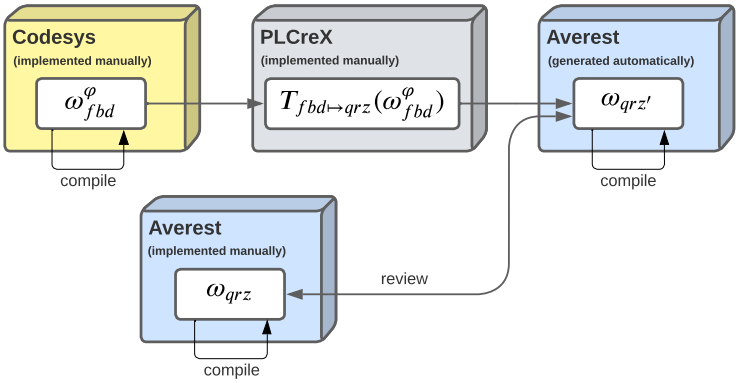


Figure 6.5.: *Test strategy to evaluate the FBD-to-Quartz transformation*

tween manually implemented and automatically generated *Quartz* models), with the following warnings:

- **Initializing input variables**: Similar to the experimental results in the context of the transformation from ST to *Quartz* (see Section 4.3), according to Lemma 4.2 with remarks in Section 6.2.4, input variables cannot be set to specific values, which is why the example *simple calculation* throws a warning for initialized input variables, since the initialization is skipped during translation.

Based on the experimental results in Table 6.1, it can be concluded that the introduced translation functions are applicable and lead to correct *Quartz* models. They can be reused in model-based design, if a few conditions are considered. These are summarized in the following section.

6.4. Summary

This chapter introduced the transformation of FBDs to *Quartz* models. For this purpose, individual translation functions were defined that take into account the sequential execution order of operators and invoked models. The applicability of these translation functions was demonstrated using a set of FBD examples. Based on the presented lemmas and experimental results, the following theorem encapsulates the entire transformation:

Theorem 6.1 (FBD-to-Quartz Translation). *Let $\omega^{\varphi}_{fbd} \in \Omega^{\varphi}_{fbd}$ be an FBD of variant $\varphi \in \{fb, fun, prg\}$, and let $T_{fbd \mapsto qrz}(\omega^{\varphi}_{fbd})$ be the model transformation of ω^{φ}_{fbd} to $\omega_{qrz'}$ using the translation functions defined in this chapter. Then, the resulting* Quartz *model $\omega_{qrz'}$:*

1. *Conforms to the syntax rules of ω_{qrz}*

Table 6.1.: *Set of FBDs and test results to evaluate the applicability of the introduced FBD-to-Quartz transformation*

Model	Source	ω_{fbd}^{φ}	$\omega_{qrz'}$	Result
2-of-3 logic function	[Sch19][1]	E.1	F.1	passed
Air Condition Control	[Tap15]	E.2	F.2	passed
Alarm function	[Sch19][1]	E.3	F.3	passed
Antivalence	[Kar18]	E.4	F.4	passed
Arithmetic operators	self	E.5	F.5	passed
Bending Machine Control	[AG20]	E.6	F.6	passed
Boolean operators	self	E.7	F.7	passed
Cylinder Control System	[Sch14]	E.8	F.8	passed
Data types and fields	self	E.9	F.9	passed
Debounce	[GDV14]	E.10	F.10	passed
Dice Numbers Indicator	[Kar18]	E.11	F.11	passed
KV Diagram optimized Chart	[Kar18]	E.12	F.12	passed
Left detection	[Sch19][1]	E.13	F.13	passed
Pollutant Indicator	[Bub17]	E.14	F.14	passed
Reservoirs Control System 1	[WZ07]	E.15	F.15	passed
Reservoirs Control System 2	[Kar18]	E.16	F.16	passed
Roll Down Shutters	[AG20]	E.17	F.17	passed
Cable Winch	[Tap15]	E.18	F.18	passed
Seven Segment Display	[WZ07]	E.19	F.19	passed
Shop Window Lighting	[AG20]	E.20	F.20	passed
Silo Valve Control System	[WZ07]	E.21	F.21	passed
Simple calculation	[GDV14]	E.22	F.22	passed with warnings
Simple Program 1	self	E.23	F.23	passed
Simple Program 2	self	E.24	F.24	passed
Smoke Detection System	[Tap15]	E.25	F.25	passed
Sports Hall Lighting	[AG20]	E.26	F.26	passed
Thermometer Code System	[Bub17]	E.27	F.27	passed
Toggle Switch 4x	[Bub17]	E.28	F.28	passed
Ventilation Control System	[Kar18]	E.29	F.29	passed
Wind Direction Indicator	[Bub17]	E.30	F.30	passed

[1]example is given as ST model and manually implemented as FBD

2. *Preserves the semantics of* ω_{fbd}^{φ}
3. *Contains constructs corresponding to the constructs of* ω_{fbd}^{φ} *and preserves the intended functionality of* ω_{fbd}^{φ} *under the following conditions:*
   - $\varphi \in \{fb, fun, prg\}$
   - $\Delta_{idcl}(\omega_{fbd}^{\varphi}) = \Delta_{in}(\omega_{fbd}^{\varphi}) \cup \Delta_{out}(\omega_{fbd}^{\varphi}) \cup \Delta_{inout}(\omega_{fbd}^{\varphi})$, *where mod-*

els are always invoked with defined input values, so no initializations are required for $\Delta_{in}(\omega_{st}^{\varphi})$

- $\Delta_{vdcl}(\omega_{fbd}^{\varphi}) = \Delta_{local}(\omega_{fbd}^{\varphi}) \cup \Delta_{inst}(\omega_{fbd}^{\varphi})$, where variables of $\Delta_{local}(\omega_{fbd}^{\varphi})$ are to be represented as memorized variables
- Imported models are available as Quartz models
- $\forall \alpha^{[+]}(\omega_{fbd}^{\varphi}) : \alpha^{[+]} \in \{\alpha_{bv}^{bool}, \alpha_{bv}^{byte}, \alpha_{bv}^{word}, \alpha_{i}^{int}, \alpha_{i}^{dint}, \alpha_{i}^{uint}, \alpha_{i}^{udint}, \alpha_{dur}, \alpha^{+}\}$, where α_{dur} can be treated as an unbounded integer and is specified in milliseconds
- $\forall \tau(\omega_{fbd}^{\varphi}) : \tau \in \{\tau_{misc}^{cst}, \tau_{misc}^{id}, \tau_{misc}^{\pi,\eta,\lambda}, \tau_{misc}^{br}, \tau_{misc}^{true}, \tau_{misc}^{false}, \tau_{misc}^{arr}, \tau_{misc}^{inv}, \tau_{comp}^{eq}, \tau_{comp}^{ne}, \tau_{comp}^{gt}, \tau_{comp}^{ge}, \tau_{comp}^{lt}, \tau_{comp}^{le}, \tau_{arith}^{mul}, \tau_{arith}^{div}, \tau_{arith}^{add}, \tau_{arith}^{sub}, \tau_{arith}^{expt}, \tau_{arith}^{mod}, \tau_{arith}^{um}, \tau_{cond}^{sel}\}$
- $\forall \sigma(\omega_{st}^{\varphi}) : \sigma \in \{\sigma_{inv}^{\vartheta}, \sigma_{ass}^{\vartheta}\}$

Proof The validity of Theorem 6.1 is proved as follows, taking into account the remarks in this chapter, when the linked lemmas of the ST-to-Quartz transformation are applied to FBDs (see Section 6.2.2, 6.2.3, 6.2.4, 6.2.5, 6.2.6, 6.2.7, 6.2.8, and 6.2.9):

1. **Syntax Conformance**: Lemma 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 6.1, 4.13, 4.14 4.15, 4.16, 4.20, and 6.1 demonstrate that each translated construct conforms to the syntax rules of ω_{qrz} as specified in Section 3.3.
2. **Semantic Preservation**: The following lemmas address the preservation of semantics for their respective constructs.
   - Model declaration: Lemma 4.1
   - Interfaces: Lemma 4.2
   - Variables: Lemma 4.6
   - Data types and fields: Lemma 4.10
   - POU imports: Lemma 4.11
   - Expressions: Lemma 4.12
   - POU invocations: Lemma 6.1, 4.13 and 4.14
   - Assignments: Lemma 4.15 and 4.16
   - Sequences: Lemma 4.20
3. **Construct Correspondence**: Given the conditions of the theorem, the provided definitions, proofs, and experimental results, it can be concluded that the translation functions produce corresponding constructs in ω_{qrz} for the considered constructs in ω_{fbd}^{φ}, preserving the original functionality.

Overall, this results in the following solutions to the challenges summarized in Section 6.2.

1. **Cyclic execution of *Quartz* models (with and without memory))**, **Event-driven execution of synchronous parallel threads**, **Sequential execution of synchronous parallel threads**, and **Dynamic system time**: The solutions follow from the summarized solutions in Section 4.4 in the context of the ST-to-Quartz transformation, since they also hold for FBDs
2. **Translation of FBD language constructs**: The solution follows from Theorem 6.1.

Chapter 7

Model Transformation of FBDs to Data-Flow Oriented SCCharts

Contents

The fourth approach to reusing existing POUs in model-based design is the transformation of FBDs into data-flow oriented SCCharts, where the goal is to create a robust set of translation functions that ensure semantic preservation during the transition. In addition to the approaches presented in [WS22; WS23; WS24b], it considers the following additional issues:

- **Model Declaration**: Mimicking the termination behavior of the initial model, i.e., distinguishing between models with and without memory
- **Interfaces and Variables**: Additional interfaces for event-driven execution control and an external controlled time (provided as a bounded integer)
- **Data Types and Fields**: Additional IEC 61131-3 data types

- **Sequential Execution**: Sequential execution via additional event-driven interfaces, which replaces the `pre` expression applied in [WS22][1]
- **POU invocations**: Instantiation and invocation of user-defined models, taking into account their individual termination behavior

The correctness of the translation functions is proved by theoretical reasoning, which includes a detailed analysis of the resulting syntax and semantics compared to the syntax rules and semantics specified in Chapter 3. In addition, the theoretical results are evaluated with real and self-defined FBDs.

This chapter is structured as follows: Section 7.1 introduces the high-level design flow and translation strategy. Section 7.2 defines the translation functions and theoretical analysis. Section 7.3 presents an evaluation of the theoretical results, and Section 7.4 summarizes the transformation.

7.1. High-Level Design Flow – FBD-to-SCChart

The high-level design flow for transforming an FBD $\omega_{fbd}^{\varphi} \in \Omega_{fbd}^{\varphi}$ of the variant *function block* ($\varphi = fb$), *function* ($\varphi = fun$), or *program* ($\varphi = prg$) to a data-flow oriented SCCharts $\omega_{scd} \in \Omega_{scd}$ is shown in Figure 7.1.

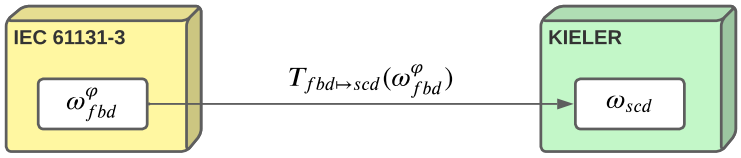


Figure 7.1.: *High-level design flow of the FBD-to-SCChart transformation*

The FBD-to-SCChart transformation $T_{fbd \mapsto scd}(\omega_{fbd}^{\varphi})$ includes the following transformation steps:

1. $t_{fbd \mapsto scd}^{\delta_\omega}(\delta_\omega(\omega_{fbd}^{\varphi}))$: Model declaration (see Section 7.2.1)
2. $t_{fbd \mapsto scd}^{\Delta_{idcl}}(\omega_{fbd}^{\varphi})$: Interfaces (see Section 7.2.2)
3. $t_{fbd \mapsto scd}^{\Delta_{vdcl}}(\omega_{fbd}^{\varphi})$: Variables (see Section 7.2.3)
4. $t_{fbd \mapsto scd}^{\alpha}(\alpha^{[+]}(\omega_{fbd}^{\varphi}))$: Data types and fields (see Section 7.2.4)
5. $t_{fbd \mapsto scd}^{\Delta_{imports}}(\omega_{fbd}^{\varphi})$: POU imports (see Section 7.2.5)
6. $t_{fbd \mapsto scd}^{\tau}(\tau(\omega_{fbd}^{\varphi}))$: Expressions (see Section 7.2.6)
7. $t_{fbd \mapsto scd}^{\Sigma_{inv}}(\sigma_{inv}^{\vartheta}(\omega_{fbd}^{\varphi}))$: POU invocations (see Section 7.2.7)
8. $t_{fbd \mapsto scd}^{\Sigma_{ass}}(\sigma_{ass}^{\vartheta}(\omega_{fbd}^{\varphi}))$: Assignments (see Section 7.2.8)
9. $t_{fbd \mapsto scd}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{scd'}))$: Sequences (see Section 7.2.9)

[1] The `pre` expression was applied in [WS22] to reflect the sequential order of assignments in the underlying FBD.

Translation Strategy:

This chapter introduces the translation strategies illustrated in Figure 7.2. Figure 7.2a shows both the initial high-level runtime behavior of an example FBD without memory and the resulting high-level runtime of the resulting SCChart. Basically, when the model is invoked at time t_i (FBD) or in *macro step* S_i (SCChart) via input `EI`, respectively, inputs `IN` are read and variables are initialized, i.e., in the resulting SCChart, the affected variables are set to their default values. The model can read a dynamic system time `CLK` during an iteration in *macro step* $S_i, \ldots, S_m$. The resulting SCChart contains a finite number of parallel threads that are executed according to the SCMoC, where the execution order of the initial FBD is enforced. During an iteration, only `CLK` is allowed to be updated externally. The final outputs `OUT` are allowed to be processed externally when an iteration terminates in *macro step* S_m when output `EO` is triggered. In contrast, Figure 7.2b shows both the initial high-level runtime behavior of an example FBD with memory and the high-level runtime of the resulting data-flow oriented SCChart. Basically, the variables are initialized at time t_0 (FBD) or in *macro step* S_0 (SCChart), respectively. The subsequent processing is equivalent to the first strategy without memory.

Challenges:

From this, the following challenges for translating FBDs to data-flow oriented SCCharts can be derived:

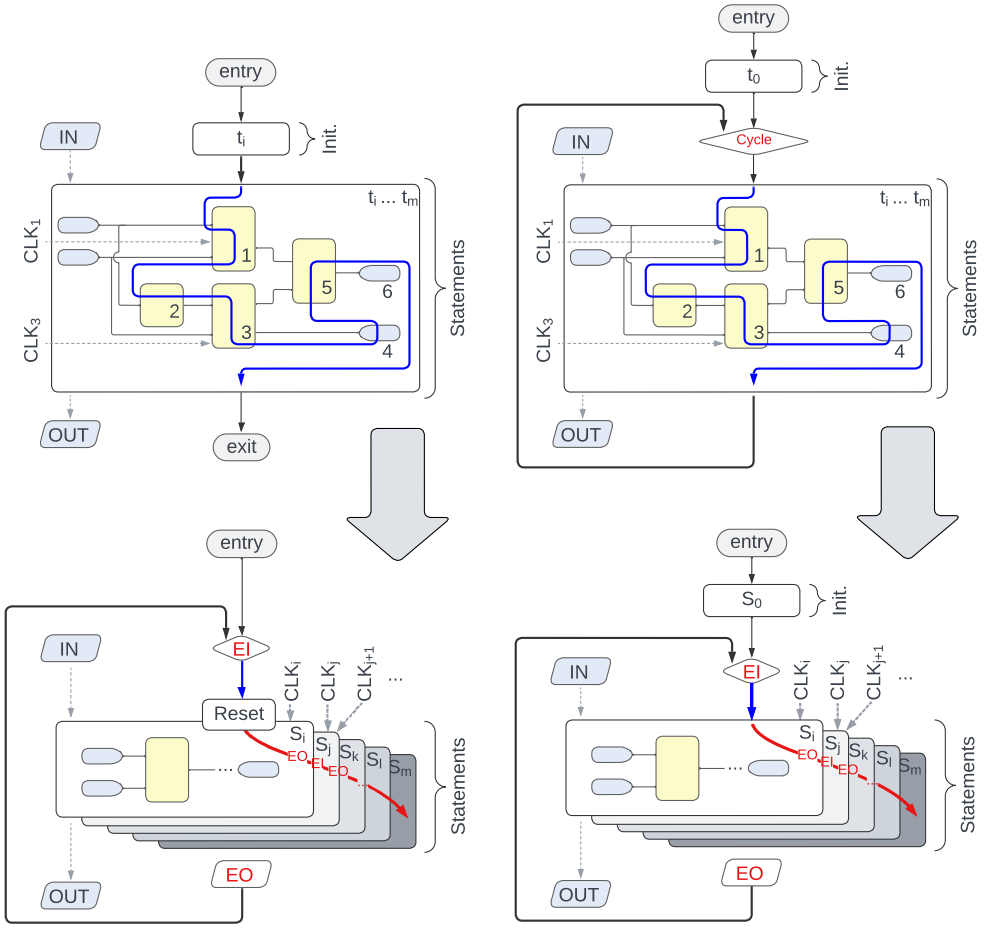
1. Cyclic execution of SCCharts (with and without memory)
2. Event-driven execution of synchronous parallel threads
3. Sequential execution of synchronous parallel threads
4. Dynamic system time
5. Translation of FBD language constructs

7.2. From FBDs to Data-Flow Oriented SCCharts

This section defines the individual translation functions for translating an FBD $\omega_{fbd}^{\varphi} \in \Omega_{fbd}^{\varphi}$ to a data-flow oriented SCChart $\omega_{scd'} \in \Omega_{scd}$ and analyzes the theoretical correctness.

7.2.1. Model Declaration

This step covers the translation function for translating an FBD declaration $\delta_{\omega}(\omega_{fbd}^{\varphi})$ to an SCChart declaration $\delta_{\omega}(\omega_{scd'})$. According to the introduced translation strategies, the resulting SCChart of an FBD, variant $\varphi \in \{fb, prg\}$, is executed in an infinite loop without reset of variables, reflecting an FBD with memory. In contrast, the resulting SCChart of an FBD, variant $\varphi = fun$, is also executed in an infinite loop, but with variables set to their default values at the beginning of each iteration, reflecting an FBD without memory. For the reset and variable assignment, the control-flow oriented $\texttt{MOVE}_{\alpha}$ SCCharts with



(a) *without memory* **(b)** *with memory*

Figure 7.2.: *FBD-to-SCChart translation strategies: high-level runtime behavior of the initial FBD (top) and resulting SCChart (bottom)*

$\alpha \in \{bool, float, int\}$ are integrated, which are derived from the IEC 61131-3 `MOVE` selection function [GDV14] and listed in Listing K.1, K.2, and K.3. These blocks simply move the input value to the output (with memory) when the `EI` port is triggered (an indicated by the `EO` output), which is illustrated in Section 7.2.8.

Definition 7.1 (Model Declaration – FBD-to-SCChart). *Let* $\Omega^{\varphi}_{fbd} = \{\omega^{\varphi}_{fbd} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible FBDs.* $\delta_{\omega}(\omega^{\varphi}_{fbd})$ *is translated to* $\delta_{\omega}(\omega_{scd'})$ *using the translation function* $t^{\delta_{\omega}}_{fbd \mapsto scd}(\delta_{\omega}(\omega^{\varphi}_{fbd}))$*, which is described by Algorithm 18.*

Correctness

To check the correctness of Definition 7.1, the following lemma is used.

Algorithm 18 Translate model declaration – FBD-to-SCChart

Input: $\delta_\omega(\omega^{\varphi}_{fbd})$
Output: $\delta_\omega(\omega_{scd'})$
Translation Function $t^{\delta_\omega}_{fbd\mapsto scd}(\delta_\omega(\omega^{\varphi}_{fbd}))$**:**

$$\delta_\omega(\omega_{scd'}) \leftarrow \left\{ \begin{array}{l} \texttt{import "MOVE}_\alpha\texttt{.sctx" ...} \\ t^{\Delta_{imports}}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd}) \\ \texttt{scchart } a_n(\omega^{\varphi}_{fbd})\{ \\ \quad \texttt{input bool EI} \\ \quad \texttt{output bool EO} \\ \quad t^{\Delta_{idcl}}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd}) \\ \quad t^{\Delta_{vdcl}}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd}) \\ \quad \texttt{ref MOVE}_\alpha \texttt{ MOVE\_01 ...} \\ \quad \texttt{dataflow:} \\ \quad\quad \texttt{MOVE\_01 = \{EI, ... \};} \\ \quad\quad t^{\Sigma}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd}) \\ \quad\quad \texttt{EO = ...;} \\ \} \end{array} \right\}$$

▷ *data-flow starts with reset to defaults using* `MOVE` *SCCharts if* $\varphi = fun$

Lemma 7.1. *Let* ω^{φ}_{fbd} *be translated to* $\omega_{scd'}$ *and* $\varphi \in \{fb, fun, prg\}$. *Then,* $t^{\delta_\omega}_{fbd\mapsto scd}(\delta_\omega(\omega^{\varphi}_{fbd}))$ *translates* $\delta_\omega(\omega^{\varphi}_{fbd})$ *to* $\delta_\omega(\omega_{scd'})$ *as specified in Definition 7.1.* $\delta_\omega(\omega_{scd'})$ *conforms to the syntax rules of* $\delta_\omega(\omega_{scd})$ *and preserves the semantics of* $\delta_\omega(\omega^{\varphi}_{fbd})$ *regarding its termination behavior.*

Proof The validity of Lemma 7.1 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\delta_\omega(\omega_{scd'})$ with the syntax rules of $\delta_\omega(\omega_{scd})$. The correctness in both cases $\varphi = fun$ and $\varphi \in \{fb, prg\}$ follows from the syntactically correct import of a `MOVE` SCChart, declaration of `EI` and `EO`, assignments, and invocations as specified in Section 3.5. Second, there are two cases to distinguish when checking the semantic correctness:

- **Case 1 ($\varphi = fun$):** $\delta_\omega(\omega_{scd'})$ starts each cycle by resetting variables to their defaults, mimicking POUs without memory. Thus, $[\![\delta_\omega(\omega_{scd})]\!]_\xi$ preserves $[\![\delta_\omega(\omega^{\varphi}_{fbd})]\!]_\xi$ regarding its termination behavior (see Section 3.2 and 3.5).
- **Case 2 ($\varphi \in \{fb, prg\}$):** $\delta_\omega(\omega_{scd'})$ starts each cycle with an invocation or assignment without resetting variables to their defaults, mimicking POUs with memory. Thus, $[\![\delta_\omega(\omega_{scd})]\!]_\xi$ preserves $[\![\delta_\omega(\omega^{\varphi}_{fbd})]\!]_\xi$ regarding its termination behavior (see Section 3.2 and 3.5).

Illustrative Example for Lemma 7.1

As simple examples, Figure 7.3 and Figure 7.4 show two SCCharts, with the graphical data-flow oriented SCCharts in the front and the compiled control-flow oriented SCCharts in the background (illustrating the resulting parallel threads). In particular, Figure 7.3 shows an SCChart with reset[2] of variables and Figure 7.4 shows an SCChart without reset of variables. The graphical views illustrate the sequential execution from invocation by `EI` to triggering `EO`. Both models increment `X1` by `1`, where the resulting SCChart with resets always returns `X1=4` (analogous to an FBD of the variant $\varphi = fun$), and `X1` in the resulting SCChart with memory either retains its current value (if `EI=false`) or is incremented by one (if `EI=true`), which is equivalent to the behavior of an FBD of the variant $\varphi \in \{fb, prg\}$. Both scenarios are evaluated within *CODESYS* and equivalent IEC 61131-3 FBDs.

```
1  import "MOVE_int.sctx"
2  scchart SIMPLE_ADD_FUN{
3    input bool EI
4    output bool EO
5    int X1 = 3
6    ref MOVE_int MOVE_01
7    ref MOVE_int MOVE_02
8
9    dataflow:
10     MOVE_01 = {EI, 3};
11     X1 = MOVE_01.OUT;
12     MOVE_02 = {MOVE_01.EO, X1
           + 1};
13     X1 = MOVE_02.OUT;
14     EO = MOVE_02.EO;
15 }
```

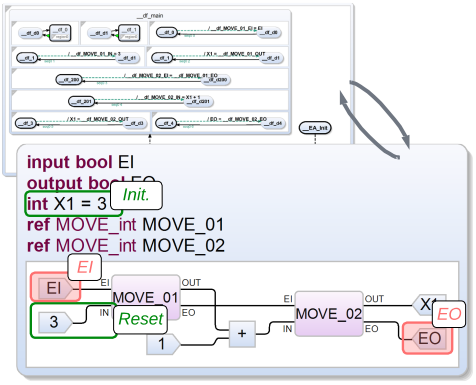


Figure 7.3.: *Example SCChart illustrating a resulting model with reset*

```
1  import "MOVE_int.sctx"
2  scchart SIMPLE_ADD_FB{
3    input bool EI
4    output bool EO
5    int X1 = 3
6    ref MOVE_int MOVE_01
7
8    dataflow:
9      MOVE_01 = {EI, X1 + 1};
10     X1 = MOVE_01.OUT;
11     EO = MOVE_01.EO;
12 }
```

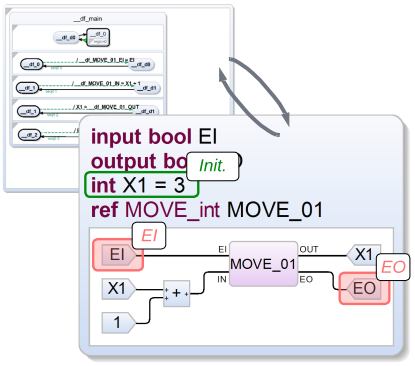


Figure 7.4.: *Example SCChart illustrating the resulting model without reset*

[2]Setting a variable to its default value requires a `MOVE` block, for which the data type must be considered.

7.2.2. Interfaces

This step covers the translation function for translating FBD interfaces $\Delta_{idcl}(\omega^{\varphi}_{fbd})$ to SCChart interfaces $\Delta_{idcl}(\omega_{scd'})$, where $\Delta_{idcl}(\omega^{\varphi}_{fbd}) = \Delta_{in}(\omega^{\varphi}_{fbd}) \cup \Delta_{out}(\omega^{\varphi}_{fbd}) \cup \Delta_{inout}(\omega^{\varphi}_{fbd})$.

Definition 7.2 (Interfaces – FBD-to-SCChart). *Let $\Omega^{\varphi}_{fbd} = \{\omega^{\varphi}_{fbd} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible FBD elements. $\Delta_{idcl}(\omega_{scd'})$ is derived from ω^{φ}_{fbd} and extended by interfaces for event-driven execution control and by an optional system time, using the translation function $t^{\Delta_{idcl}}_{fbd \mapsto scd}(\omega^{\varphi}_{fbd})$, which is described by Algorithm 19.*

Correctness

To check the correctness of Definition 7.2, the following lemmas are used.

Lemma 7.2. *Let ω^{φ}_{fbd} be translated to $\omega_{scd'}$ and $\varphi \in \{fb, prg\}$. Then, for each interface $e_i \in \mathcal{E}_i(\omega^{\varphi}_{fbd})$, $t^{\Delta_{idcl}}_{fbd \mapsto scd}(\omega^{\varphi}_{fbd})$ extends $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, or $\Delta_{inout}(\omega_{scd'})$ as specified in Definition 7.2. $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, and $\Delta_{inout}(\omega_{scd'})$ conform to the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, and $\Delta_{inout}(\omega_{scd})$ regarding the storage class, data type, and name. With and without initialization, $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, and $\Delta_{inout}(\omega_{scd'})$ preserves the semantics of $\Delta_{in}(\omega^{\varphi}_{fbd})$, $\Delta_{out}(\omega^{\varphi}_{fbd})$, and $\Delta_{inout}(\omega^{\varphi}_{fbd})$ regarding information flow, modifiability, and initialization.*

Proof The validity of Lemma 7.2 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, and $\Delta_{inout}(\omega_{scd'})$ with the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, and $\Delta_{inout}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of added interfaces $\mathcal{E}_i(\omega^{\varphi}_{fbd})$:

1. **Base Case:** When $\mathcal{E}_i(\omega^{\varphi}_{fbd}) = \varnothing$, there are no input variables, output variables, and inout variables to add, which trivially conforms to the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, and $\Delta_{inout}(\omega_{scd})$, since these sets remain unchanged.
2. **Induction Hypothesis:** The lemma holds for any set of input variables, output variables, and inout variables.
3. **Inductive Step:** Adding an element to input variables, output variables, and inout variables results in an additional element in $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, and $\Delta_{inout}(\omega_{scd'})$. Their syntax still conforms to the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, and $\Delta_{inout}(\omega_{scd})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{in}(\omega_{scd})]\!]_{\xi}$, $[\![\Delta_{out}(\omega_{scd})]\!]_{\xi}$, and $[\![\Delta_{inout}(\omega_{scd})]\!]_{\xi}$ (see Section 3.5) with $[\![\Delta_{in}(\omega^{\varphi}_{fbd})]\!]_{\xi}$, $[\![\Delta_{out}(\omega^{\varphi}_{fbd})]\!]_{\xi}$, and $[\![\Delta_{inout}(\omega^{\varphi}_{fbd})]\!]_{\xi}$ (see Section 3.2).

Algorithm 19 Translate interfaces – FBD-to-SCChart

Input: ω_{fbd}^{φ}
Output: $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, $\Delta_{inout}(\omega_{scd'})$, $\Sigma_{seq}(\omega_{scd'})$
Translation Function $t_{fbd\mapsto scd}^{\Delta_{idcl}}(\omega_{fbd}^{\varphi})$**:**

- $\Delta_{in}(\omega_{scd'})$ ← add `input bool EI;` ▷ *add Boolean input variable*
- $\Delta_{out}(\omega_{scd'})$ ← add `output bool EO;` ▷ *add Boolean output variable*
- **if** ω_{fbd}^{φ} *contains time-based logic* **then**
  - $\Delta_{in}(\omega_{scd'})$ ← add `input int CLK;` ▷ *add input variable*
- **end**
- **if** $\varphi = fun \wedge e_{rT} \neq \emptyset$, *where* $e_{rT} \in \mathcal{E}_{rT}(\omega_{fbd}^{\varphi}), \mathcal{E}_{rT}(\omega_{fbd}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{fbd}^{\varphi})$ **then**
  - $\Delta_{out}(\omega_{scd'})$ ← add `output` $t_{fbd\mapsto scd}^{\alpha}(\alpha(e_{rT}))\ a_n(\omega_{fbd}^{\varphi})$; ▷ *add output variable*
- **end**
- **forall** $e_i \in \mathcal{E}_{i}(\omega_{fbd}^{\varphi})$ **do**
  - **if** $e_i = e_{iVs}$, *where* $e_{iVs} \in \mathcal{E}_{iVs}(\omega_{fbd}^{\varphi}), \mathcal{E}_{iVs}(\omega_{fbd}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{fbd}^{\varphi})$ **then**
    - $\Delta_{in}(\omega_{scd'})$ ← add `input` $t_{fbd\mapsto scd}^{\alpha}(\alpha(e_i))\ a_n(e_i)$ `[=` $t_{fbd\mapsto scd}^{\tau_{misc}}(\pi)$`]`; ▷ *add input variable with optional initialization*
  - **end**
  - **if** $e_i = e_{oVs}$, *where* $e_{oVs} \in \mathcal{E}_{oVs}(\omega_{fbd}^{\varphi}), \mathcal{E}_{oVs}(\omega_{fbd}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{fbd}^{\varphi})$ **then**
    - $\Delta_{out}(\omega_{scd'})$ ← add `output` $t_{fbd\mapsto scd}^{\alpha}(\alpha(e_i))\ a_n(e_i)$ `[=` $t_{fbd\mapsto scd}^{\tau_{misc}}(\pi)$`]`; ▷ *add output variable with optional initialization*
    - **if** $\varphi = fun$ **then**
      - $\Sigma_{seq}(\omega_{scd'}) \leftarrow \left\{ \begin{array}{l} \texttt{MOVE}_{iN}\ \texttt{= \{EI,}\ t_{fbd\mapsto scd}^{\tau_{misc}}(\pi)\texttt{\};} \\ a_n(e_i)\ \texttt{= MOVE}_{iN}\texttt{.OUT;} \end{array} \right\}$ ▷ *add reset to default value with corresponding `MOVE` instance*
    - **end**
  - **end**
  - **if** $e_i = e_{iOVs}$, *where* $e_{iOVs} \in \mathcal{E}_{iOVs}(\omega_{fbd}^{\varphi}), \mathcal{E}_{iOVs}(\omega_{fbd}^{\varphi}) \subset \mathcal{E}_{i}(\omega_{fbd}^{\varphi})$ **then**
    - $\Delta_{inout}(\omega_{scd'})$ ← add `input output` $t_{fbd\mapsto scd}^{\alpha}(\alpha(e_i))\ a_n(e_i)$ `[=` $t_{fbd\mapsto scd}^{\tau_{misc}}(\pi)$`]`; ▷ *add inout variable with optional initialization*
  - **end**
- **end**

Illustrative Example for Lemma 7.2

As an example, below are some derived interfaces of the `FBD_AIR_COND_CTRL` model[3]:

```
1    input bool IN1                    // added to Δ_in(ω_scd')
2    input bool IN2                    // added to Δ_in(ω_scd')
3    ...
```

[3]Both models, FBD and SCChart, are included in Appendix E.2 and G.2.

Lemma 7.3. *Let ω_{fbd}^{φ} be translated to $\omega_{scd'}$ and $\varphi = fun$. Then, for each interface $e_i \in \mathcal{E}_i(\omega_{fbd}^{\varphi})$, $t_{fbd\mapsto scd}^{\Delta_{idcl}}(\omega_{fbd}^{\varphi})$ extends $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$ including optional initialization, or $\Delta_{inout}(\omega_{scd'})$, and adds possible assignments to defaults to $\Sigma_{seq}(\omega_{scd'})$ as specified in Definition 7.2. $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, and $\Delta_{inout}(\omega_{scd'})$ conform to the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, and $\Delta_{inout}(\omega_{scd})$ regarding the storage class, data type, and name. $\Sigma_{seq}(\omega_{scd'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{scd})$. With and without assignments to defaults, $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, and $\Delta_{inout}(\omega_{scd'})$ in combination with $\Sigma_{seq}(\omega_{scd'})$ preserves the semantics of $\Delta_{in}(\omega_{st}^{\varphi})$, $\Delta_{out}(\omega_{st}^{\varphi})$, and $\Delta_{inout}(\omega_{st}^{\varphi})$ regarding information flow, modifiability, and initialization.*

Proof The validity of Lemma 7.3 is checked as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, $\Delta_{inout}(\omega_{scd'})$, and $\Sigma_{seq}(\omega_{scd'})$ with the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, $\Delta_{inout}(\omega_{scd})$, and $\Sigma_{seq}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of added interfaces $\mathcal{E}_i(\omega_{fbd}^{\varphi})$:

1. **Base Case:** When $\mathcal{E}_i(\omega_{fbd}^{\varphi}) = \varnothing$, there are no input variables, output variables with possible initialization, and inout variables, and thus no statements to add, which trivially conforms to the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, $\Delta_{inout}(\omega_{scd})$, $\Sigma_{inv}^{fb}(\omega_{scd})$, and $\Sigma_{ass}^{imm}(\omega_{scd})$, since these sets remain unchanged and are optional.
2. **Induction Hypothesis:** The lemma holds for any set of input variables, output variables, and inout variables.
3. **Inductive Step:** Adding an element to input variables, output variables, and inout variables results in an additional element in $\Delta_{in}(\omega_{scd'})$, $\Delta_{out}(\omega_{scd'})$, $\Delta_{inout}(\omega_{scd'})$, $\Sigma_{inv}^{fb}(\omega_{scd})$, and $\Sigma_{ass}^{imm}(\omega_{scd})$. Their syntax still conforms to the syntax rules of $\Delta_{in}(\omega_{scd})$, $\Delta_{out}(\omega_{scd})$, $\Delta_{inout}(\omega_{scd})$, $\Sigma_{inv}^{fb}(\omega_{scd})$, and $\Sigma_{ass}^{imm}(\omega_{scd})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{in}(\omega_{scd})]\!]_{\xi}$, $[\![\Delta_{out}(\omega_{scd})]\!]_{\xi}$, and $[\![\Delta_{inout}(\omega_{scd})]\!]_{\xi}$ in combination with the SOS transition rules of $\Sigma_{inv}^{fb}(\omega_{scd})$ and $\Sigma_{ass}^{imm}(\omega_{scd})$, noting that `MOVE` instances terminate immediately (see Section 3.5) with $[\![\Delta_{in}(\omega_{fbd}^{\varphi})]\!]_{\xi}$, $[\![\Delta_{out}(\omega_{fbd}^{\varphi})]\!]_{\xi}$, and $[\![\Delta_{inout}(\omega_{fbd}^{\varphi})]\!]_{\xi}$ (see Section 3.2).

Illustrative Example for Lemma 7.3

As an example, below are some derived interfaces of the `FBD_TWO_OF_THREE` model[4]:

[4]Both models, FBD and SCChart, are included in Appendix E.1 and G.1.

```
1     input bool xB1_Temp              // added to Δin(ωscd′)
2     input bool xB2_Temp              // added to Δin(ωscd′)
3     ...
```

Lemma 7.4. *Let ω_{fbd}^{φ} be translated to $\omega_{scd'}$ and $\varphi \in \{fb, fun, prg\}$. Then, $t_{fbd \mapsto scd}^{\Delta_{idcl}}(\omega_{fbd}^{\varphi})$ adds the additional input `EI` to $\Delta_{in}(\omega_{scd'})$ and the additional output `EO` to $\Delta_{out}(\omega_{scd'})$ as specified in Definition 7.2. $\Delta_{in}(\omega_{scd'})$ and $\Delta_{out}(\omega_{scd'})$ conform to the syntax rules of $\Delta_{in}(\omega_{scd})$ and $\Delta_{out}(\omega_{scd})$.*

Proof The validity of Lemma 7.4 is proved by comparing the resulting syntax of $\Delta_{in}(\omega_{scd'})$ and $\Delta_{out}(\omega_{scd'})$ with the syntax rules of $\Delta_{in}(\omega_{scd})$ and $\Delta_{out}(\omega_{scd})$ as specified in see Section 3.5.

Illustrative Example for Lemma 7.4

As an example, below are the derived interfaces of the `SIMPLE_ADD_FUN` model introduced in Lemma 7.1:

```
1     input bool EI                    // added to Δin(ωscd′)
2     output bool EO                   // added to Δout(ωscd′)
```

Lemma 7.5. *Let ω_{fbd}^{φ} be translated to $\omega_{scd'}$, $\varphi \in \{fb, fun, prg\}$, and ω_{fbd}^{φ} contains time-based logic, where time is a readable variable and is synchronized externally. Then, $t_{fbd \mapsto scd}^{\Delta_{idcl}}(\omega_{fbd}^{\varphi})$ adds an additional input to $\Delta_{in}(\omega_{scd'})$ as specified in Definition 7.2. $\Delta_{in}(\omega_{scd'})$ conforms to the syntax rules of $\Delta_{in}(\omega_{scd})$.*

Proof The validity of Lemma 7.5 is proved by comparing the resulting syntax of $\Delta_{in}(\omega_{scd'})$ with the syntax rules of $\Delta_{in}(\omega_{scd})$ as specified in Section 3.5.

Illustrative Example for Lemma 7.5

As an example, below is the derived input of the `FBD_DEBOUNCE` model[5]:

```
1     input int CLK                    // added to Δin(ωscd′)
```

Lemma 7.6. *Let ω_{fbd}^{fun} be translated to $\omega_{scd'}$ and ω_{fbd}^{fun} has a specified return type, which is processed by ω_{fbd}^{fun}. Then, $t_{fbd \mapsto scd}^{\Delta_{idcl}}(\omega_{fbd}^{\varphi})$ adds an additional memorized output for the specified return type to $\Delta_{out}(\omega_{scd'})$ as specified in Definition 7.2. $\Delta_{out}(\omega_{scd'})$ conforms to the syntax rules of $\Delta_{out}(\omega_{scd})$.*

[5] Both models, FBD and SCChart, are included in Appendix E.10 and G.10.

Proof The validity of Lemma 7.6 is proved by comparing the resulting syntax of $\Delta_{out}(\omega_{scd'})$ with the syntax rules of $\Delta_{out}(\omega_{scd})$ as specified in Section 3.5.

Illustrative Example for Lemma 7.6

As an example, below is the derived output of the `FBD_SIMPLE_FUN` model[6]:

```
1    output float FBD_SIMPLE_FUN      // added to Δout(ωscd')
```

7.2.3. Variables

This step covers the translation function for translating local FBD variables $\Delta_{vdcl}(\omega^{\varphi}_{fbd})$ to local SCChart variables $\Delta_{vdcl}(\omega_{scd'})$, where $\Delta_{vdcl}(\omega^{\varphi}_{fbd}) = \Delta_{local}(\omega^{\varphi}_{fbd}) \cup \Delta_{inst}(\omega^{\varphi}_{fbd})$.

Definition 7.3 (Variables – FBD-to-SCChart). *Let $\Omega^{\varphi}_{fbd} = \{\omega^{\varphi}_{fbd} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible FBD elements. $\Delta_{vdcl}(\omega_{scd'})$ is derived from ω^{φ}_{fbd} using the translation function $t^{\Delta_{vdcl}}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd})$, which is described by Algorithm 20.*

Algorithm 20 Translate variables – FBD-to-SCChart

Input: ω^{φ}_{fbd}
Output: $\Delta_{local}(\omega_{scd'})$, $\Delta_{inst}(\omega_{scd'})$, $\Sigma_{seq}(\omega_{scd'})$
Translation Function $t^{\Delta_{vdcl}}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd})$**:**

- **forall** $e_{lVs} \in \mathcal{E}_i(\omega^{\varphi}_{fbd})$ **do**
  - **if** $e_d(e_{lVs}) = \varnothing$ **then**
    - $\Delta_{local}(\omega_{scd'}) \leftarrow$ add $t^{\alpha}_{fbd\mapsto scd}(\alpha(e_{lVs}))\ \ a_n(e_{lVs})$ `[=` $t^{\tau_{misc}}_{fbd\mapsto scd}(\pi)$`]`;
      ▷ *add local variable with optional initialization*
    - **if** $\varphi = fun$ **then**
      - $\Sigma_{seq}(\omega_{scd'}) \leftarrow \left\{ \begin{array}{l} \texttt{MOVE}_{iN}\ \texttt{=}\ \{\texttt{EI,}\ t^{\tau_{misc}}_{fbd\mapsto scd}(\pi)\}\texttt{;} \\ a_n(e_i)\ \texttt{=}\ \texttt{MOVE}_{iN}\texttt{.OUT;} \end{array} \right\}$
        ▷ *add reset to default value with corresponding `MOVE` instance*
    - **end**
  - **else**
    - $\Delta_{inst}(\omega_{scd'}) \leftarrow$ add `ref` $e_d(e_{lVs})\ \ a_n(e_{lVs})$;
      ▷ *add local variable for instance*
  - **end**
- **end**

Correctness

To check the correctness of Definition 7.3, the following lemmas are used.

[6] Both models, FBD and SCChart, are included in Appendix E.22 and G.22.

Lemma 7.7. *Let ω_{fbd}^{φ} be translated to $\omega_{scd'}$ and $\varphi \in \{fb, prg\}$. Then, for each local variable $e_{lVs} \in \mathcal{E}_i(\omega_{fbd}^{\varphi})$ that is not derived from external models $e_d(e_{lVs}) = \varnothing$, $t_{fbd \mapsto scd}^{\Delta_{vdcl}}(\omega_{fbd}^{\varphi})$ adds a local variable including optional initialization to $\Delta_{local}(\omega_{scd'})$ as specified in Definition 7.3. $\Delta_{local}(\omega_{scd'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{scd})$ regarding the storage class, data type, and name. With and without initialization, $\Delta_{local}(\omega_{scd'})$ preserves the semantics of $\Delta_{local}(\omega_{fbd}^{\varphi})$ regarding modifiability and initialization.*

Proof The validity of Lemma 7.7 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{local}(\omega_{scd'})$ with the syntax rules of $\Delta_{local}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of added variables $\mathcal{E}_i(\omega_{fbd}^{\varphi})$:

1. **Base Case:** When $\mathcal{E}_i(\omega_{fbd}^{\varphi}) = \varnothing$, there are no local variables and thus no statements to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{scd})$, since this set remains unchanged and is optional.
2. **Induction Hypothesis:** The lemma holds for any set of local variables with possible initialization.
3. **Inductive Step:** Adding an element to local variables with initialization results in an additional element in $\Delta_{local}(\omega_{scd'})$. Its syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{scd})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{local}(\omega_{scd})]\!]_{\xi}$ (see Section 3.5) with $[\![\Delta_{local}(\omega_{fbd}^{\varphi})]\!]_{\xi}$ (see Section 3.2).

Illustrative Example for Lemma 7.7

As an example, below is a snippet of the derived local variables of the `FBD_DATATYPES` model[7]:

```
1    bool A1                  // added to Δlocal(ωscd')
2    bool A2 = true           // added to Δlocal(ωscd')
```

Lemma 7.8. *Let ω_{fbd}^{φ} be translated to $\omega_{scd'}$ and $\varphi \in \{fun\}$. Then, for each local variable $e_{lVs} \in \mathcal{E}_i(\omega_{fbd}^{\varphi})$ that is not derived from external models $e_d(e_{lVs}) = \varnothing$, $t_{fbd \mapsto scd}^{\Delta_{vdcl}}(\omega_{fbd}^{\varphi})$ adds a local variable including optional initialization to $\Delta_{local}(\omega_{scd'})$, and adds possible assignments to defaults to $\Sigma_{seq}(\omega_{scd'})$ as specified in Definition 7.3. $\Delta_{local}(\omega_{scd'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{scd})$ regarding the storage class, data type, and name. $\Sigma_{seq}(\omega_{scd'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{scd})$. With and*

[7] Both models, FBD and SCChart, are included in Appendix E.9 and G.9.

without initialization, $\Delta_{local}(\omega_{scd'})$ in combination with $\Sigma_{seq}(\omega_{scd'})$ preserves the semantics of $\Delta_{local}(\omega^{\varphi}_{fbd})$ regarding modifiability and initialization.

Proof The validity of Lemma 7.8 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{local}(\omega_{scd'})$ and $\Sigma_{seq}(\omega_{scd'})$ with the syntax rules of $\Delta_{local}(\omega_{scd})$ and $\Sigma_{seq}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of added variables $\mathcal{E}_i(\omega^{\varphi}_{fbd})$:

1. **Base Case:** When $\mathcal{E}_i(\omega^{\varphi}_{fbd}) = \varnothing$, there are no local variables and thus no statements to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{scd})$, $\Sigma^{fb}_{inv}(\omega_{scd})$, and $\Sigma^{imm}_{ass}(\omega_{scd})$, since these sets remain unchanged and are optional.
2. **Induction Hypothesis:** The lemma holds for any set of local variables.
3. **Inductive Step:** Adding an element to local variables results in an additional element in $\Delta_{local}(\omega_{scd'})$, $\Sigma^{fb}_{inv}(\omega_{scd})$, and $\Sigma^{imm}_{ass}(\omega_{scd})$. Their syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{scd})$, $\Sigma^{fb}_{inv}(\omega_{scd})$, and $\Sigma^{imm}_{ass}(\omega_{scd})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{local}(\omega_{scd})]\!]_{\xi}$ in combination with the SOS transition rules of $\Sigma^{fb}_{inv}(\omega_{scd})$ and $\Sigma^{imm}_{ass}(\omega_{scd})$, noting that `MOVE` instances terminate immediately (see Section 3.5) with $[\![\Delta_{local}(\omega^{\varphi}_{fbd})]\!]_{\xi}$ (see Section 3.2).

Illustrative Example for Lemma 7.8

As an example, below are a derived variable including reset of the `SIMPLE_ADD_FUN` model introduced in Lemma 7.1:

```
1    int X1 = 3                          // added to Δin(ωscd')
2    ...
3    dataflow:
4      MOVE_01 = {EI, 3};                // added to Σseq(ωscd')
5      X1 = MOVE_01.OUT;                 // added to Σseq(ωscd')
6      ...
```

Lemma 7.9. *Let ω^{φ}_{fbd} be translated to $\omega_{scd'}$ and $\varphi \in \{fb, prg\}$. Then, for each local variable $e_{lVs} \in \mathcal{E}_i(\omega^{\varphi}_{fbd})$ that is derived from external models $e_d(e_{lVs}) \neq \varnothing$, $t^{\Delta_{vdcl}}_{fbd \mapsto scd}(\omega^{\varphi}_{fbd})$ adds a local variable to $\Delta_{inst}(\omega_{scd'})$ as specified in Definition 7.3. $\Delta_{inst}(\omega_{scd'})$ conforms to the syntax rules of $\Delta_{inst}(\omega_{scd})$ regarding the name and preserves the semantics of $\Delta_{inst}(\omega^{\varphi}_{fbd})$ regarding usage within the model.*

Proof The validity of Lemma 7.9 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{inst}(\omega_{scd'})$ with the syntax rules of $\Delta_{inst}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of added variables $\mathcal{E}_i(\omega^{\varphi}_{fbd})$:

1. **Base Case:** When $\mathcal{E}_i(\omega^{\varphi}_{fbd}) = \varnothing$, there are no instances, which trivially conforms to the syntax rules of $\Delta_{inst}(\omega_{scd})$, since this set remains unchanged and is optional.
2. **Induction Hypothesis:** The lemma holds for any set of instances.
3. **Inductive Step:** Adding an instance results in an additional element in $\Delta_{inst}(\omega_{scd'})$. Its syntax still conforms to the syntax rules of $\Delta_{inst}(\omega_{scd})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{inst}(\omega_{scd})]\!]_{\xi}$ (see Section 3.5) with $[\![\Delta_{inst}(\omega^{\varphi}_{fbd})]\!]_{\xi}$ (see Section 3.2).

Illustrative Example for Lemma 7.9

As an example, below is a snippet of the derived local variables of the `FBD_AIR_COND_CTRL` model[8]:

```
1     ref RS RS0                // added to Δ_inst(ω_scd')
```

7.2.4. Data Types and Fields

This step covers the translation function for translating FBD data types and fields $\mathcal{A}^{[+]}(\omega^{\varphi}_{fbd})$ to SCChart data types and fields $\mathcal{A}^{[+]}(\omega_{scd'})$. FBD data types and fields correspond to ST model data types and fields [GDV14]. Furthermore, SCChart data types and fields correspond to SCL data types and fields (see Section 3.5). Consequently, the translation strategy introduced in Section 5.2.4 can be applied to the FBD-to-SCChart transformation, resulting in $t^{\alpha}_{fbd\mapsto scd}(\alpha^{[+]}(\omega^{\varphi}_{fbd})) \coloneqq t^{\alpha}_{st\mapsto scl}(\alpha^{[+]}(\omega^{\varphi}_{fbd}))$. Due to the equivalence to the ST-to-SCL transformation, no additional examples for the FBD-to-SCChart transformation are included at this point.

7.2.5. POU Imports

This step covers the translation function for translating POU imports $\Delta_{imports}(\omega^{\varphi}_{fbd})$ to SCChart imports $\Delta_{imports}(\omega_{scd'})$.

Definition 7.4 (POU imports – FBD-to-SCChart). *Let $\Omega^{\varphi}_{fbd} = \{\omega^{\varphi}_{fbd} \mid \varphi \in \{fb, fun, prg\}\}$ be the set of possible FBD elements. POU imports in FBDs $\Delta_{imports}(\omega^{\varphi}_{fbd})$ are derived from external blocks $e_{lVs} \in \mathcal{E}(\omega^{\varphi}_{fbd})$, $e_d(e_{lVs}) \neq \varnothing$ and user-defined blocks $e_{bfun'} \in F'(\omega^{\varphi}_{fbd})$. The imports are translated to SCChart imports $\Delta_{imports}(\omega_{scd'})$ and extended by data type specific `MOVE` models using the translation function $t^{\Delta_{imports}}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd})$, which is described by Algorithm 21.*

[8] Both models, FBD and SCChart, are included in Appendix E.2 and G.2.

Algorithm 21 Translate POU imports – FBD-to-SCChart

Input: ω_{fbd}^{φ}
Output: $\Delta_{imports}(\omega_{scd'})$
Translation Function $t_{fbd\mapsto scd}^{\Delta_{imports}}(\omega_{fbd}^{\varphi})$**:**
  forall $\alpha(lhs(\sigma_{ass}^{imm}(\omega_{fbd}^{\varphi})))$ **do**
    $\Delta_{imports}(\omega_{scd'}) \leftarrow$ add `import "MOVE_(`$\alpha(lhs(\sigma_{ass}^{imm}(\omega_{fbd}^{\varphi}))))$`.sctx";`
      ▷ *data type is derived from left-hand side of assigned variables*
  end
  forall $e_{lVs} \in \mathcal{E}(\omega_{fbd}^{\varphi}), e_d(e_{lVs}) \neq \varnothing$ **do**
    $\Delta_{imports}(\omega_{scd'}) \leftarrow$ add `import "(`$e_d(e_{lVs})$`).sctx";`
  end
  forall $e_{bfun'} \in F'(\omega_{fbd}^{\varphi})$ *(derived from* ω_{fbd}^{φ}*)* **do**
    $\Delta_{imports}(\omega_{scd'}) \leftarrow$ add `import "(`$a_n(e_{bfun'})$`).sctx";`
  end

Correctness

To check the correctness of Definition 7.4, the following lemmas are used.

Lemma 7.10. *Let the IEC 61131-3 standard function blocks (`RS`, `SR`, `TOF`, `TON`) [GDV14] be available as semantically and syntactically correct SCChart models. Then, for each instance* $e_{lVs} \in \mathcal{E}(\omega_{fbd}^{\varphi}), e_d(e_{lVs}) \neq \varnothing$ *and user-defined function* $e_{bfun'} \in F'(\omega_{fbd}^{\varphi})$, $t_{fbd\mapsto scd}^{\Delta_{imports}}(\omega_{fbd}^{\varphi})$ *adds the corresponding import to* $\Delta_{imports}(\omega_{scd'})$ *(if not already imported) as specified in Definition 7.4.* $\Delta_{imports}(\omega_{scd'})$ *conforms to the syntax rules of* $\Delta_{imports}(\omega_{scd})$ *and preserves the semantics of* $\Delta_{imports}(\omega_{fbd}^{\varphi})$ *regarding instantiation and usage.*

Proof The validity of Lemma 7.10 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{imports}(\omega_{scd'})$ with the syntax rules of $\Delta_{imports}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of instances $\mathcal{E}(\omega_{fbd}^{\varphi}) = \{e_{lVs} \mid e_d(e_{lVs}) \neq \varnothing\}$ and user-defined functions $F'(\omega_{fbd}^{\varphi})$:

1. **Base Case:** When $\mathcal{E}(\omega_{fbd}^{\varphi}) = \varnothing$ and $F'(\omega_{fbd}^{\varphi}) = \varnothing$, there are no modules to import, which trivially conforms to the syntax rules of $\Delta_{imports}(\omega_{scd})$, since this set remains unchanged and is optional.
2. **Induction Hypothesis:** The lemma holds for any set of instances and any set of user-defined functions.
3. **Inductive Step:** Adding an instance and user-defined function to $\mathcal{E}(\omega_{fbd}^{\varphi})$ and $F'(\omega_{fbd}^{\varphi})$ results in two additional elements in $\Delta_{imports}(\omega_{scd'})$ (one for the instance and one for the user-defined function). Its syntax still conforms to the syntax rules of $\Delta_{imports}(\omega_{scd})$.

Second, the semantic correctness is checked by comparing $[\![\Delta_{imports}(\omega_{scd})]\!]_\xi$ (see Section 3.5) with $[\![\Delta_{imports}(\omega^{\varphi}_{fbd})]\!]_\xi$ (see Section 3.2).

Illustrative Example for Lemma 7.10

As an example, below are the derived import of the `FBD_AIR_COND_CTRL` model[9]:

```
1    import "RS.sctx"            // added to Δ_imports(ω_scd')
```

Lemma 7.11. *Let the data types of required `MOVE` models be derived from the assigned variables* $\alpha(lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd})))$. *Then, for each data type,* $t^{\Delta_{imports}}_{fbd\mapsto scd}(\omega^{\varphi}_{fbd})$ *adds the corresponding import to* $\Delta_{imports}(\omega_{scd'})$ *(if not already imported) as specified in Definition 7.4.* $\Delta_{imports}(\omega_{scd'})$ *conforms to the syntax rules of* $\Delta_{imports}(\omega_{scd})$.

Proof The validity of Lemma 7.11 is proved by comparing the resulting syntax of $\Delta_{imports}(\omega_{scd'})$ with the syntax rules of $\Delta_{imports}(\omega_{scd})$ as specified in Section 3.5 using induction on the data types of the left-hand side of the assigned variables $\alpha(lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd})))$:

1. **Base Case:** When $lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd})) = \varnothing$, there are no derived data types, which trivially conforms to the syntax rules of $\Delta_{imports}(\omega_{scd})$, since this set remains unchanged and is optional.
2. **Induction Hypothesis:** The lemma holds for any set of instances and any set of user-defined functions.
3. **Inductive Step:** Adding an instance and user-defined function to $lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd}))$ results in an additional element in $\Delta_{imports}(\omega_{scd'})$. Its syntax still conforms to the syntax rules of $\Delta_{imports}(\omega_{scd})$.

Illustrative Example for Lemma 7.11

As an example, below is the derived import of the `SIMPLE_ADD_FUN` model introduced in Lemma 7.1:

```
1    import "MOVE_int.sctx"      // added to Δ_imports(ω_scd')
```

7.2.6. Expressions

This step covers the translation function for translating expressions in FBDs $\mathcal{T}(\omega^{\varphi}_{fbd})$ to expressions in SCCharts $\mathcal{T}(\omega_{scd'})$. The translation strategy for the ST-to-SCL transformation introduced in Section 5.2.5 can be applied to the FBD-to-SCChart transformation, because the translations are linked to

[9] Both models, FBD and SCChart, are included in Appendix E.2 and G.2.

the individual specifications of the ST and FBD constructs, and the resulting expressions for the SCL models are the same as for SCCharts (see Section 3.5), resulting in $t^{\tau}_{fbd \mapsto scd}(\tau(\omega^{\varphi}_{fbd})) \coloneqq t^{\tau}_{st \mapsto scl}(\tau(\omega^{\varphi}_{fbd}))$. As summarized in Lemma 5.8, only a subset of the considered FBD expressions can be translated to equivalent SCChart expressions (using internal operators).

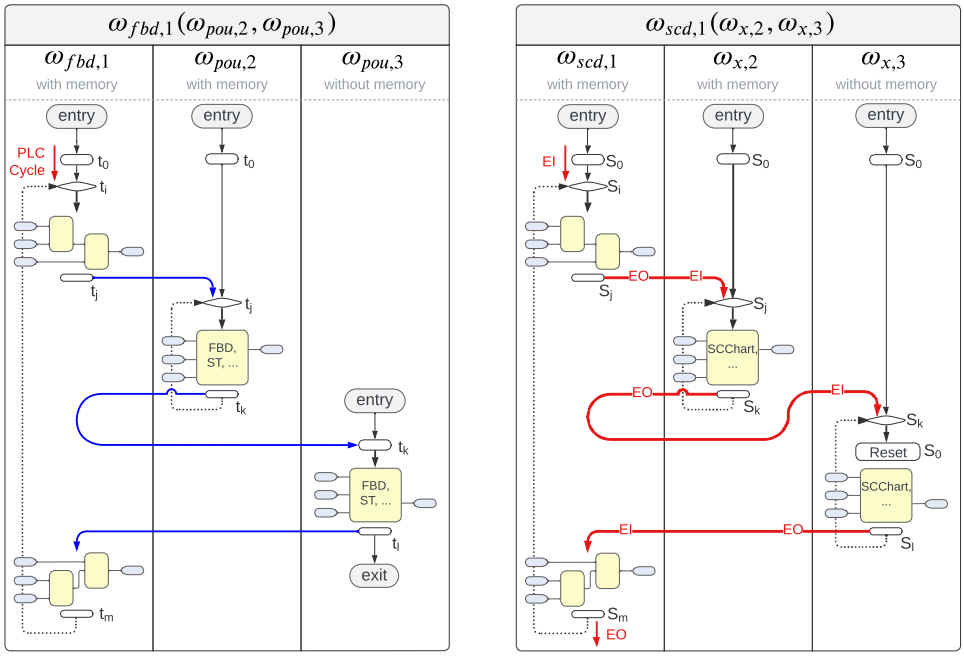
7.2.7. POU Invocations

This step covers the translation function for translating POU invocations in FBDs $\Sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})$ to model invocations in SCCharts $\Sigma^{\vartheta}_{inv}(\omega_{scd'})$. As an illustration, Figure 7.5 shows the high-level runtime behavior of an example FBD with memory $\omega_{fbd,1}$ invoking a model with memory $\omega_{pou,2}$ and a model without memory $\omega_{pou,3}$, as well as the resulting SCChart $\omega_{scd,1}$ that invokes the two models, $\omega_{x,2}$ and $\omega_{x,3}$. In particular, Figure 7.5a shows that $\omega_{fbd,1}$ is triggered in every PLC cycle at time t_i and invokes $\omega_{pou,2}$ and $\omega_{pou,3}$ depending on their execution order at time t_j and t_k, until an iteration of $\omega_{fbd,1}$ terminates at time t_m, with $i < j < k < l < m$ and $i, j, k, l, m \geq 0$. In contrast, Figure 7.5b shows that an iteration of the resulting SCChart $\omega_{scd,1}$ is triggered in each PLC cycle at *macro step* S_i and invokes $\omega_{x,2}$ and $\omega_{x,3}$ depending on their execution order at *macro step* S_j and S_k. The models are invoked by the additional interfaces `EI` and `EO` of the corresponding components. Consequently, an iteration of $\omega_{scd,1}$ terminates at *macro step* S_m and allows multiple *macro steps* of instances due to the event-driven execution control using `EI` and `EO`. If an iteration of an instance terminates within the same *macro step* per PLC cycle, it is possible to force a sequential execution of parallel threads of SCCharts using the sequential statement (see Section 3.5) without additional interfaces for event-driven execution control. However, for a generic application of the introduced approach, this thesis focuses on event-driven execution control. It is worth noting that this event-driven execution control focuses on model invocations, although in principle it is possible to invoke any operator of depending on its execution order (which was pursued, for example, in an approach to transform IEC 61131-3 models into IEC 61499 models [Wen+09b]).

Definition 7.5 (POU Invocations – FBD-to-SCChart). *Let* $\Omega^{\varphi}_{fbd} = \{\omega^{\varphi}_{fbd} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible FBD elements. A POU invocation* $\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}) \in \Sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})$ *(considering additional* `MOVE` *instances for resetting variables) is translated to a model invocation* $\sigma^{\vartheta}_{inv}(\omega_{scd'}) \in \Sigma^{\vartheta}_{inv}(\omega_{scd'})$ *in SCCharts using the translation function* $t^{\Sigma_{inv}}_{fbd \mapsto scd}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))$*, which is described by Algorithm 22 (that applies function* `get_expr` *described by Algorithm 17).*

Correctness

To check the correctness of Definition 7.5, the following lemma is used.



(a) *Initial FBD* **(b)** *Resulting SCChart*

Figure 7.5.: *High-level runtime behavior of a model with memory invoking two models (Approach: FBD-to-SCChart)*

Algorithm 22 Invoke POU – FBD-to-SCChart

Input: $\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})$

Output: $\sigma^{fb}_{inv}(\omega_{scd'})$

Translation Function $t^{\Sigma_{inv}}_{fbd\mapsto scd}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))$**:**

$\sigma^{fb}_{inv}(\omega_{scd'}) \leftarrow a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))$ `= {MOVE_(`ID_{n-1}`).EO, [CLK,] `$\mathcal{I}$`};`

▷ *add invocation with* `EO` *trigger of previous component* ID_{n-1} *and system time (if specified)*

forall $i \in \mathcal{I}, \mathcal{I} = \mathcal{E}_{iVs}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})) \cup \mathcal{E}_{iOVs}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))$ **do**

$i \mapsto t^{\tau}_{fbd\mapsto scd}$`(get_expr(`$i$`));`

▷ *add translated input or inout argument*

end

Lemma 7.12. *Let* ω^{φ}_{fbd} *be translated to* $\omega_{scd'}$ *and* $\varphi \in \{fb, fun, prg\}$*. Then,* $t^{\Sigma_{inv}}_{fbd\mapsto scd}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))$ *translates a model invocation in FBDs* $\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})$ *to a model invocation in SCCharts* $\sigma^{fb}_{inv}(\omega_{scd'})$ *with related arguments including synchronized system time (if specified) as specified in Definition 7.5.* $\sigma^{fb}_{inv}(\omega_{scd'})$ *conforms to the syntax rules of* $\sigma^{fb}_{inv}(\omega_{scd})$ *and preserves the SOS rules of* $\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})$ *regarding termination behavior.*

Proof The validity of Lemma 7.12 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma^{fb}_{inv}(\omega_{scd'})$ with the syntax rules of $\sigma^{fb}_{inv}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of input variables including inout variables $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))) \cup \mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})))$ of an invoked instance $\sigma^{fb}_{inv}(\omega^{\varphi}_{fbd})$ with system time:

1. **Base Case:** When $\sigma^{fb}_{inv}(\omega^{\varphi}_{fbd}) \neq \varnothing$, $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))) = \varnothing$, and $\mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd}))) = \varnothing$, there are no existing interfaces to add, which conforms to the syntax rules of $\sigma^{fb}_{inv}(\omega_{scd})$.
2. **Induction Hypothesis:** The lemma holds for any set of input variables including inout variables of an invoked instance.
3. **Inductive Step:** Adding an input and inout variable to $\mathcal{E}_{iVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})))$ and $\mathcal{E}_{iOVs}(a_{iN}(\sigma^{\vartheta}_{inv}(\omega^{\varphi}_{fbd})))$ results in two additional interfaces and $\sigma^{fb}_{inv}(\omega_{scd'})$, which conforms to the syntax rules of $\sigma^{fb}_{inv}(\omega_{scd})$. Syntactic correctness of `get_expr` follows from Lemma 6.1.

Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma^{fb}_{inv}(\omega_{scd})$ (see Section 3.5) with the SOS transition rules of $\sigma^{fb}_{inv}(\omega^{\varphi}_{fbd})$ (see Section 3.2).

Illustrative Example for Lemma 7.12

As an example, below are the derived invocations of the `FBD_DEBOUNCE` model[10]. Furthermore, Figure 7.6 shows the graphical SCChart (compiled by *KIELER*), which illustrates the sequential execution of the instances.

```
...
dataflow:
  // DB_ON.EI ⇐ EI
  DB_ON = {EI, CLK, IN, DB_TIME}
  // DB_OFF.EI ⇐ DB_ON.EO
  DB_OFF = {DB_ON.EO, CLK, !(IN), DB_TIME}
  // MOVE_01.EI ⇐ DB_OFF.EO
  MOVE_01 = {DB_OFF.EO, DB_OFF.ET}
  ...
  // DB_FF.EI ⇐ MOVE_01.EO
  DB_FF = {MOVE_01.EO, DB_ON.Q, DB_OFF.Q}
  // MOVE_02.EI ⇐ DB_FF.EO
  MOVE_02 = {DB_FF.EO, DB_FF.Q1}
  ...
```

7.2.8. Assignments

This step covers the translation function for translating assignments in FBDs $\Sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd})$ to assignments in SCCharts $\Sigma^{\vartheta}_{ass}(\omega_{scd'})$ or the corresponding sequence $\Sigma_{seq}(\omega_{scd'})$, respectively. The strategy is to place a `MOVE` block between each variable assignment to ensure that only the final output value of the previous block is processed, i.e., when the previous block terminates. This is

[10] Both models, FBD and SCChart, are included in Appendix E.10 and G.10.

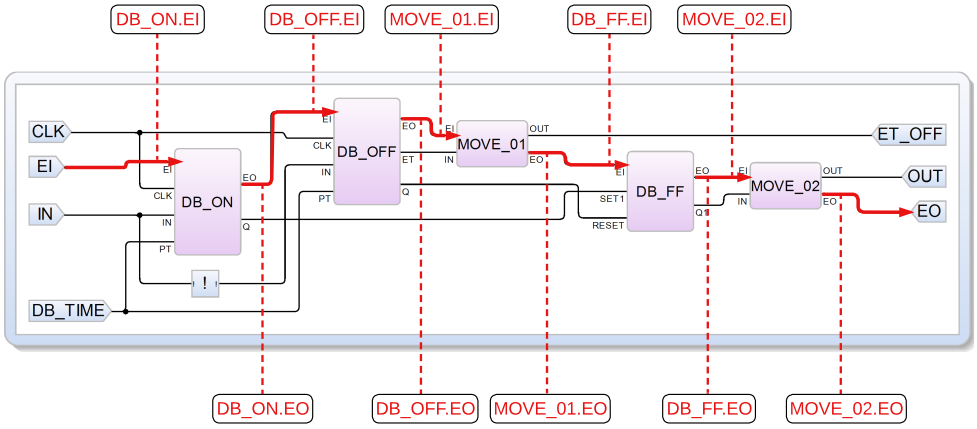


Figure 7.6.: *Graphical SCChart of the translated* `FBD_DEBOUNCE` *model*

illustrated in Figure 7.7 using the `FBD_SIMPLE_PRG2` example, which instantiates the `ST_LOOP_HEAD` example[11] mentioned in Section 5.3. In particular, Figure 7.7a shows, that the intermediate output value is not assigned to `OUT`, because `MOVE_01` and `ST_LOOP_HEAD0` have not terminated. In contrast, Figure 7.7b shows, that the final output value is assigned to `OUT`, as soon as `MOVE_01` and `ST_LOOP_HEAD0` terminate. This reflects the runtime behavior of the initial FBD shown in Figure 7.7c and Figure 7.7d in both scenarios, within and after the first PLC cycle.

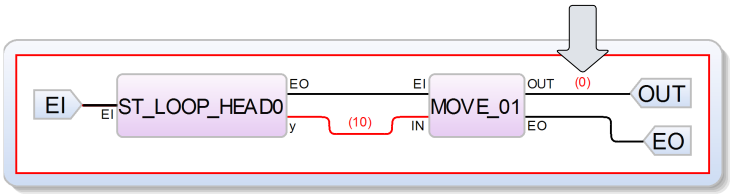
Definition 7.6 (Translation of assignments – FBD-to-SCChart). *Let* $\Omega^{\varphi}_{fbd} = \{\omega^{\varphi}_{fbd} \mid \varphi \in \{fb, prg\}\}$ *be the set of possible FBD elements and* $\Sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}) = \{\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}) \mid lhs(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd})) \in \mathcal{E}_{oV}(\omega^{\varphi}_{fbd}) \cup \mathcal{E}_{iOV}(\omega^{\varphi}_{fbd})\}$ *be the set of assigned variables, where* $(\mathcal{E}_{oV}(\omega^{\varphi}_{fbd}) \cup \mathcal{E}_{iOV}(\omega^{\varphi}_{fbd})) \cap \mathcal{E}_{iV}(\omega^{\varphi}_{fbd})$ *is not necessarily an empty set. An immediate assignment* $\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd}) \in \Sigma^{imm}_{ass}(\omega^{\varphi}_{fbd})$ *(i.e.,* $rhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd}))$ *does not depend on* $lhs(\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd}))$ *and a delayed assignment* $\sigma^{del}_{ass}(\omega^{\varphi}_{fbd}) \in \Sigma^{del}_{ass}(\omega^{\varphi}_{fbd})$ *(i.e.,* $rhs(\sigma^{del}_{ass}(\omega^{\varphi}_{fbd}))$ *in FBDs does depend on* $lhs(\sigma^{del}_{ass}(\omega^{\varphi}_{fbd}))$ *is translated to a sequence* $\Sigma_{seq}(\omega_{scd'})$ *in SCCharts using the translation function* $t^{\Sigma_{ass}}_{fbd \mapsto scd}(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}))$*, which is described by Algorithm 23.*

Correctness

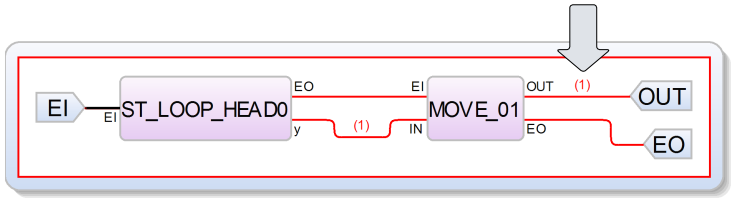
To check the correctness of Definition 7.6, the following lemma is used.

Lemma 7.13. *Let* ω^{φ}_{fbd} *be translated to* $\omega_{scd'}$*,* $\varphi \in \{fb, fun, prg\}$*, and* $\vartheta = imm$*. Then,* $t^{\Sigma_{ass}}_{fbd \mapsto scd}(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}))$ *translates* $\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd})$ *to* $\Sigma_{seq}(\omega_{scd'})$ *as specified in Definition 7.6.* $\Sigma_{seq}(\omega_{scd'})$ *conforms to the syntax rules*

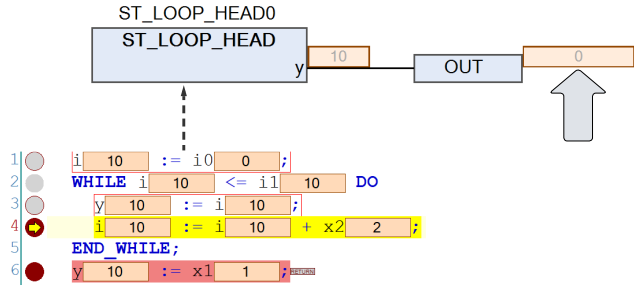
[11] The translated SCL model was compiled to an equivalent control-flow oriented SCChart using *KIELER*.



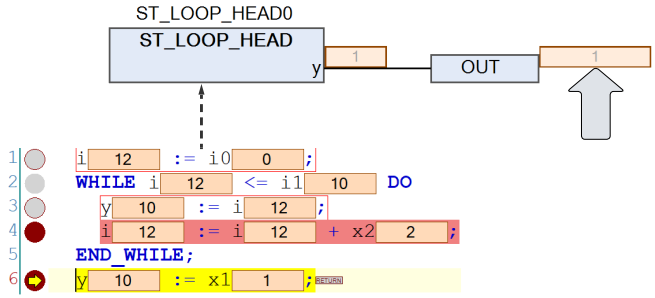
(a) *View in* KIELER *(after the 6th* macro step*, i.e., within the first PLC cycle)*



(b) *View in* KIELER *(after the 7th* macro step*, i.e., after the first PLC cycle)*



(c) *View in* CODESYS *(after the 6th simulation step, i.e., within the first PLC cycle)*



(d) *View in* CODESYS *(after the 7th simulation step, i.e., after the first PLC cycle)*

Figure 7.7.: *Views during simulation of the* `FBD_SIMPLE_PRG2` *example*

of $\Sigma_{seq}(\omega_{scd})$*, preserves the SOS rules of* $\sigma_{ass}^{imm}(\omega_{fbd}^{\varphi})$*, and respects the SCMoC.*

Proof The validity of Lemma 7.13 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{inv}^{fb}(\omega_{scd'})$ and

Algorithm 23 Translate assignment – FBD-to-SCChart

Input: $\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd})$
Output: $\Sigma_{seq}(\omega^{\varphi}_{scd'})$
Translation Function $t^{\Sigma_{ass}}_{fbd\mapsto scd}(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}))$**:**

$$\Sigma_{seq}(\omega_{scd'}) \leftarrow \left\{ \begin{array}{l} \texttt{MOVE}_{iN} \texttt{ = \{MOVE\_(}ID_{n-1}\texttt{).EO, } t^{\tau}_{fbd\mapsto scd}\texttt{(} \\ \quad \texttt{get\_expr(} \; lhs(\sigma^{\vartheta}_{ass}(\omega^{\varphi}_{fbd}))\texttt{))\};} \\ a_n(e_i) \texttt{ = MOVE}_{iN}\texttt{.OUT;} \end{array} \right\}$$

▷ *add invocation with* `EO` *trigger of previous component* ID_{n-1}*)*

$\Sigma^{imm}_{ass}(\omega_{scd'})$ with the syntax rules of $\Sigma^{fb}_{inv}(\omega_{scd})$ and $\Sigma^{imm}_{ass}(\omega_{scd})$ as specified in Section 3.5. Second, the semantic correctness is proved by comparing the SOS transition rules of $\Sigma^{fb}_{inv}(\omega_{scd})$ and $\Sigma^{imm}_{ass}(\omega_{scd})$, noting that `MOVE` instances terminate immediately (see Section 3.5) with the SOS transition rules of $\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd})$ (see Section 3.2). The SCMoC is respected because FBDs are limited to implicit loops in this approach [GDV14].

Illustrative Example for Lemma 7.13

As an example, the immediately assigned variables of the `FBD_SIMPLE_PRG2` model are translated as follows[12]:

```
1    OUT := ST_LOOP_HEAD0.y;      // initial FBD assignment
2        ⇒ MOVE_01 = {ST_LOOP_HEAD0.EO, ST_LOOP_HEAD0.y}
3        ⇒ OUT = MOVE_01.OUT
4        ⇒ EO = MOVE_01.EO
```

7.2.9. Sequences

This step covers the translation function for translating sequences in FBDs $\Sigma_{seq}(\omega^{\varphi}_{fbd})$ to sequences in SCCharts $\Sigma_{seq}(\omega_{scd'})$.

Definition 7.7 (Translation of sequences – FBD-to-SCChart). *Let* $\Omega^{\varphi}_{fbd} = \{\omega^{\varphi}_{fbd} \mid \varphi \in \{fb, fun, prg\}\}$ *be the set of possible FBD elements. There are the following variants of FBD statements:*

- *POU invocations:* $\sigma_{inv}(\omega^{\varphi}_{fbd}) \in \Sigma_{inv}(\omega^{\varphi}_{fbd})$
- *Assignments:* $\sigma^{imm}_{ass}(\omega^{\varphi}_{fbd}) \in \Sigma^{imm}_{ass}(\omega^{\varphi}_{fbd})$, $\sigma^{del}_{ass}(\omega^{\varphi}_{fbd}) \in \Sigma^{del}_{ass}(\omega^{\varphi}_{fbd})$

According to the Definition 7.5 and 7.6, these FBD statements are translated to the following variants of SCChart statements:

- *Model invocations:* $\sigma_{inv}(\omega_{scd'}) \in \Sigma_{inv}(\omega_{scd'})$
- *Assignments:* $\sigma^{imm}_{ass}(\omega_{scd'}) \in \Sigma^{imm}_{ass}(\omega_{scd'})$

A set of the resulting SCChart statements represents a sequence, denoted as $\Sigma_{seq}(\omega_{scd'})$*. The translation function* $t^{\Sigma_{seq}}_{fbd\mapsto scd}(\Sigma_{seq}(\omega_{scd'}))$ *inserts* $\sigma_i(\omega_{scd'}) \in \Sigma_{seq}(\omega_{scd'})$ *to* $\omega_{scd'}$*, following the process described by Algorithm 24.*

[12] Both models, FBD and SCChart, are included in Appendix E.24 and G.24.

Algorithm 24 Add sequence – FBD-to-SCChart

Input: $\Sigma_{seq}(\omega_{scd'})$
Output: $\omega_{scd'}$
Translation Function $t^{\Sigma_{seq}}_{fbd \mapsto scd}(\Sigma_{seq}(\omega_{scd'}))$**:**
  forall $\sigma_i \in \Sigma_{seq}(\omega_{scd'})$ **do**
    $\omega_{scd'} \leftarrow$ add σ_i to the position w.r.t. its execution order and dependent constructs;
      ▷ *execution order of parallel threads is enforced by event-driven execution control*
  end

Correctness

To check the correctness of Definition 7.7, the following lemma is used.

Lemma 7.14. *Let ω^{φ}_{fbd} be translated to $\omega_{scd'}$, $\varphi \in \{fb, prg\}$, $\Sigma_{seq}(\omega_{scd'}) \neq \varnothing$, and $\Sigma_{seq}(\omega_{scd'})$ be syntactically correct. Then, $t^{\Sigma_{seq}}_{fbd \mapsto scd}(\Sigma_{seq}(\omega_{scd'}))$ inserts each translated statement $\sigma_i \in \Sigma_{seq}(\omega_{scd'})$ to $\omega_{scd'}$. The resulting SCChart $\omega_{scd'}$ conforms to the syntax rules of ω_{scd}, preserves the SOS rules of ω^{φ}_{fbd} (in particular with regard to the execution order of the statements), and respects the SCMoC.*

Proof The validity of Lemma 7.14 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{scd'})$ with the syntax rules of $\Sigma_{seq}(\omega_{scd})$ as specified in Section 3.5 using induction on the number of statements to be added $\Sigma_{seq}(\omega_{scd'})$:

1. **Base Case:** When $\Sigma_{seq}(\omega_{scd'}) = \varnothing$, there are no statements to be added, which conforms to the syntax rules of $\delta_{\omega}(\omega_{scd})$.
2. **Induction Hypothesis:** The lemma holds for any set of statements to be added $\Sigma_{seq}(\omega_{scd'})$.
3. **Inductive Step:** Adding a statement results in a statement to be added that conforms to the syntax rules $\Sigma_{seq}(\omega_{scd})$, because the syntactic correctness of this statement to be added has been proved in the corresponding section.

Second, the SOS transition rules and the SCMoC are respected by the statements themselves. The order is maintained by sequential execution, which is ensured by event-driven execution control.

Illustrative Example for Lemma 7.14

As an example, below are the inserted statements in the resulting SCChart of the `FBD_DEBOUNCE` model[13]:

[13] Both models, FBD and SCChart, are included in Appendix E.10 and G.10.

```
...
dataflow:
  DB_ON = {EI, CLK, IN, DB_TIME}              ⇐ σ1
  DB_OFF = {DB_ON.EO, CLK, !(IN), DB_TIME}    ⇐ σ2
  MOVE_01 = {DB_OFF.EO, DB_OFF.ET}            ⇐ σ3
  ET_OFF = MOVE_01.OUT                        ⇐ σ4
  DB_FF = {MOVE_01.EO, DB_ON.Q, DB_OFF.Q}     ⇐ σ5
  MOVE_02 = {DB_FF.EO, DB_FF.Q1}              ⇐ σ6
  OUT = MOVE_02.OUT                           ⇐ σ7
  EO = MOVE_02.EO                             ⇐ σ8
```

7.3. Experimental Results

The applicability of the introduced translation functions is evaluated with the FBDs listed in Table 7.1. For evaluation purposes, the listed FBDs and the expected SCCharts are manually implemented to verify the applicability of the isolated translation functions. To ensure the correctness of both models, FBDs and SCCharts, they are compiled with the built-in compilers of *CODESYS* and *KIELER*. The translation functions have been implemented as a prototype in *PLCreX*, assuming the FBDs are available in *PLCopen* `xml` format, resulting in the overall test strategy shown in Figure 7.8. The correctness of the resulting SCCharts is verified in two ways: (1) through manual reviews, differences between the expected SCCharts and the automatically generated SCCharts are identified, and (2) using the built-in compilers of *KIELER*, the syntactic correctness of the automatically generated SCCharts is ensured.

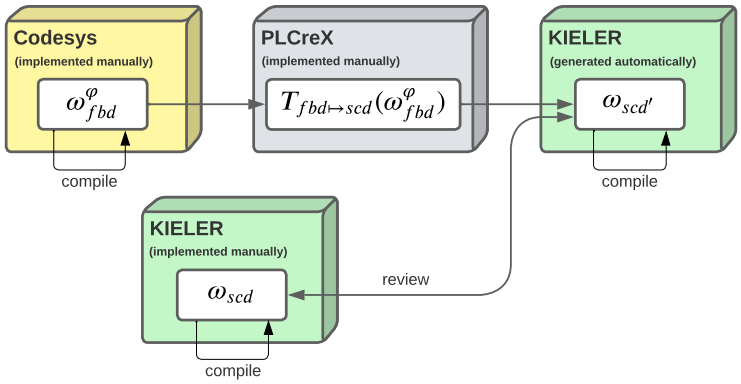


Figure 7.8.: *Test strategy to evaluate the FBD-to-SCChart transformation*

Both tests passed for all examples (ignoring minor formatting differences between manually implemented and automatically generated SCCharts), with the following warnings:

- **Supported operators**: Similar to the experimental results in the context of the transformation from ST to SCL (see Section 5.3), according to Lemma 5.8 with remarks in Section 7.2.6, only a subset of the considered FBD expressions can be translated to corresponding internal SCChart expressions. Therefore, the example *arithmetic operators* throws

Table 7.1.: *Set of FBDs and test results to evaluate the applicability of the introduced FBD-to-SCChart transformation*

Model	Source	ω_{fbd}^{φ}	$\omega_{scd'}$	Result
2-of-3 logic function	[Sch19][1]	E.1	G.1	passed
Air Condition Control	[Tap15]	E.2	G.2	passed
Alarm function	[Sch19][1]	E.3	G.3	passed
Antivalence	[Kar18]	E.4	G.4	passed
Arithmetic operators	self	E.5	G.5	passed with warnings
Bending Machine Control	[AG20]	E.6	G.6	passed
Boolean operators	self	E.7	G.7	passed
Cylinder Control System	[Sch14]	E.8	G.8	passed
Data types and fields	self	E.9	G.9	passed with warnings
Debounce	[GDV14]	E.10	G.10	passed
Dice Numbers Indicator	[Kar18]	E.11	G.11	passed
KV Diagram optimized Chart	[Kar18]	E.12	G.12	passed
Left detection	[Sch19][1]	E.13	G.13	passed
Pollutant Indicator	[Bub17]	E.14	G.14	passed
Reservoirs Control System 1	[WZ07]	E.15	G.15	passed
Reservoirs Control System 2	[Kar18]	E.16	G.16	passed
Roll Down Shutters	[AG20]	E.17	G.17	passed
Cable Winch	[Tap15]	E.18	G.18	passed
Seven Segment Display	[WZ07]	E.19	G.19	passed
Shop Window Lighting	[AG20]	E.20	G.20	passed
Silo Valve Control System	[WZ07]	E.21	G.21	passed
Simple calculation	[GDV14]	E.22	G.22	passed with warnings
Simple Program 1	self	E.23	G.23	passed
Simple Program 2	self	E.24	G.24	passed
Smoke Detection System	[Tap15]	E.25	G.25	passed
Sports Hall Lighting	[AG20]	E.26	G.26	passed
Thermometer Code System	[Bub17]	E.27	G.27	passed
Toggle Switch 4x	[Bub17]	E.28	G.28	passed
Ventilation Control System	[Kar18]	E.29	G.29	passed
Wind Direction Indicator	[Bub17]	E.30	G.30	passed

a warning for affected expressions. Affected expressions are skipped during translation.

- **Supported data types**: Similar to the experimental results in the context of the transformation from ST to SCL (see Section 5.3), according to Lemma 5.7 with remarks in Section 7.2.4, only a subset of the considered FBD data types can be translated to corresponding SCChart data types. Therefore, the example *data types and fields* throws a warning for affected declarations. Affected declarations and expressions are skipped

during translation.

- **Reset of variables**: According to Lemma 7.3, only output variables are considered to be reset to their default values for the FBD variant $\varphi = fun$. Otherwise, external values are overridden by default values due to the SCMoC. Therefore, the example *simple calculation* throws a warning for the affected input variable. The reset to default values for input and inout variables is skipped during translation.

Based on the experimental results in Table 7.1, it can be concluded that the introduced translation functions are applicable and lead to correct SCCharts that can be reused in model-based design, if a few conditions are considered. These are summarized in the following section.

7.4. Summary

This chapter introduced the transformation of FBDs to data-flow oriented SCCharts. For this purpose, individual translation functions were defined that take into account the sequential execution order of blocks and assignments. The applicability of these translation functions was demonstrated using a set of FBD examples. Based on the presented lemmas and experimental results, the following theorem encapsulates the entire transformation:

Theorem 7.1 (FBD-to-SCChart Translation). *Let $\omega_{fbd}^{\varphi} \in \Omega_{fbd}^{\varphi}$ be an FBD of the variant $\varphi \in \{fb, fun, prg\}$ and let $T_{fbd \mapsto scd}(\omega_{fbd}^{\varphi})$ be the model transformation of ω_{fbd}^{φ} to $\omega_{scd'}$ using the translation functions defined in this chapter. Then, the resulting SCChart $\omega_{scd'}$:*

1. *Conforms to the syntax rules of ω_{scd}*
2. *Preserves the semantics of ω_{fbd}^{φ}*
3. *Contains constructs corresponding to the constructs of ω_{fbd}^{φ} and preserves the intended functionality of ω_{fbd}^{φ} under the following conditions:*
   - $\varphi \in \{fb, fun, prg\}$
   - $\Delta_{idcl}(\omega_{fbd}^{\varphi}) = \Delta_{in}(\omega_{fbd}^{\varphi}) \cup \Delta_{out}(\omega_{fbd}^{\varphi}) \cup \Delta_{inout}(\omega_{fbd}^{\varphi})$, *where models are always invoked with defined input values, so no reset are required for $\Delta_{in}(\omega_{fbd}^{\varphi})$ for FBD variant $\varphi = fun$*
   - $\Delta_{vdcl}(\omega_{fbd}^{\varphi}) = \Delta_{local}(\omega_{fbd}^{\varphi}) \cup \Delta_{inst}(\omega_{fbd}^{\varphi})$
   - *Imported models are available as SCCharts*
   - $\forall \alpha^{[+]}(\omega_{fbd}^{\varphi}) : \alpha^{[+]} \in \{\alpha_{bv}^{bool}, \alpha_{i}^{int}, \alpha_{i}^{dint}, \alpha_{i}^{uint}, \alpha_{dur}, \alpha^{+}\}$, *where α_{dur} can be treated as a bounded integer and is specified in milliseconds*

- $\forall \tau(\omega_{fbd}^{\varphi}) : \tau \in \{\tau_{misc}^{cst}, \tau_{misc}^{id}, \tau_{misc}^{\pi,\eta,\lambda}, \tau_{misc}^{br}, \tau_{misc}^{true}, \tau_{misc}^{false}, \tau_{misc}^{arr}, \tau_{misc}^{inv}, \tau_{comp}^{eq}, \tau_{comp}^{ne}, \tau_{comp}^{gt}, \tau_{comp}^{ge}, \tau_{comp}^{lt}, \tau_{comp}^{le}, \tau_{arith}^{mul}, \tau_{arith}^{div}, \tau_{arith}^{add}, \tau_{arith}^{sub}, \tau_{arith}^{expt}, \tau_{arith}^{mod}, \tau_{arith}^{um}, \tau_{cond}^{sel}\}$
- $\forall \sigma(\omega_{fbd}^{\vartheta}) : \sigma \in \{\sigma_{inv}^{\vartheta}, \sigma_{ass}^{\vartheta}\}$

Proof The validity of Theorem 7.1 is proved as follows:

1. **Syntax Conformance**: Lemma 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 5.7 (see remarks in Section 7.2.4), 7.10, 7.11, 5.8 (see remarks in Section 7.2.6), 7.12, 7.13, and 7.14 demonstrate that each translated construct conforms to the syntax rules of ω_{scd} as specified in Section 3.5.
2. **Semantic Preservation**: The following lemmas address the preservation of semantics for their respective constructs.
   - Model declaration: Lemma 7.1
   - Interfaces: Lemma 7.2 and 7.3
   - Variables: Lemma 7.7, 7.8, and 7.9
   - Data types and fields: Lemma 5.7 (see remarks in Section 7.2.4)
   - POU imports: Lemma 7.10
   - Expressions: Lemma 5.8 (see remarks in Section 7.2.6)
   - POU invocations: Lemma 7.12
   - Assignments: Lemma 7.13
   - Sequences: Lemma 7.14
3. **Construct Correspondence**: Given the conditions of the theorem, the provided definitions, proofs, and experimental results, it can be concluded that the translation functions produce corresponding constructs in ω_{scd} for the considered constructs in ω_{fbd}^{φ}, preserving the original functionality.

Overall, this results in the following solutions to the challenges summarized in Section 7.1.

1. **Cyclic execution of SCCharts (with and without memory)**: The cyclic execution of a resulting SCChart (with and without memory) depends on the invocation by an external model. When an SCChart is invoked, an iteration is triggered that terminates after a finite number of *macro steps* ($n \geq 0$). Unlike variables of a resulting SCChart based on an FBD with memory, variables of a resulting SCChart based on an FBD without memory are set to their default values (if necessary) at the beginning of each iteration.

2. **Event-driven execution of synchronous parallel threads**: An iteration of an SCChart and iterations of its instances (executed in parallel threads) are triggered by additional interfaces (`EI` and `EO`) of each component, which allows event-driven execution control.
3. **Sequential execution of synchronous parallel threads**: Due to the event-driven execution control, the invoking SCChart can invoke instances depending on the feedback of the previous blocks. This allows a sequential execution of synchronous parallel threads, allowing an individual number of *macro steps* of the invoked models.
4. **Dynamic system time**: A dynamic system time is realized by an additional input `CLK`, i.e., the clock is controlled externally and processed within the resulting SCChart with read access.
5. **Translation of FBD language constructs**: The solution follows from Theorem 7.1.

Chapter 8

Formal Methods-Based Optimization of Data-Flow Models

Contents

This chapter focuses on a formal methods-based optimization of data-flow models (like data-flow oriented SCCharts, *Quartz* models, and others), where the goal is to automatically identify potentially optimizable submodels and optimize them while ensuring semantic preservation and leaving the non-modifiable components unchanged. The purpose is to reduce the number of

components in data-flow-related expressions where the user can configure the optimization strategy. More specifically, this chapter proposes a generic identification strategy of submodels in real-world applications and introduces an applicable formal methods-based optimization strategy for data-flow models using *NuSMV*, *Z3Py*, and *PLCreX*. In addition to the approaches presented in [WS23; WS24b], it considers the following additional optimization strategy:

- **Pattern-based submodel refactoring**: A pattern-based submodel refactoring of the optimized formulas, which improves the optimization results as demonstrated in Section 8.2

The correctness of the semantics during the optimization is ensured by an integrated equivalence check using *NuSMV* and *Z3Py*. In addition, the syntactic correctness is checked for the set of real-world applications of the case study by compilation after reconstruction.

This chapter is structured as follows: Section 8.1 introduces the high-level design flow and optimization strategy. Section 8.2 defines the translation and optimization steps, as well as the theoretical analysis. Section 8.3 presents an evaluation of the theoretical results and analyzes the optimization potential in real-world examples. Section 8.4 summarizes the optimization.

8.1. High-Level Design Flow – Optimization

The high-level design flow with a focus on optimizing an FBD $\omega^{\varphi}_{fbd} \in \Omega^{\varphi}_{fbd}$, a data-flow oriented ST model $\omega^{\varphi}_{st} \in \Omega^{\varphi}_{st}$, a data-flow oriented *Quartz* model $\omega_{qrz} \in \Omega_{qrz}$, and a data-flow oriented SCChart $\omega_{scd} \in \Omega_{scd}$ is shown in Figure 8.1, with $\omega^{\vartheta}_{pou} \in \{\omega^{\vartheta}_{st}, \omega^{\vartheta}_{fbd}\}$. The individual transformations $T_{pou \mapsto d}(\omega^{\varphi}_{pou})$, $T_{qrz \mapsto d}(\omega_{qrz})$, $T_{scd \mapsto d}(\omega_{scd})$, and $T_{d \mapsto d}(\mathcal{M}(\omega_d))$ are explained in the next section.

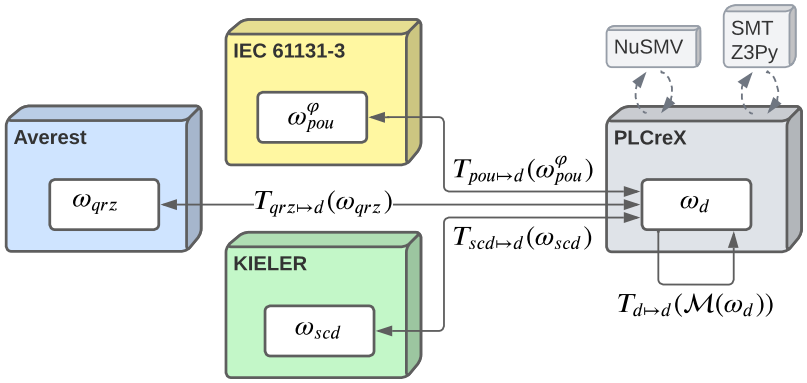


Figure 8.1.: *High-level design flow of the optimization process with focus on the models and system architecture*

Optimization Strategy:

The basic concept of the optimization process is shown in Figure 8.2. The idea is to express graphical data-flow models in textual form and then split them into two categories: (1) submodels that can be potentially optimized and (2) non-modifiable components. The submodels are then simplified using formal methods and pattern-based submodel refactoring. In particular, the model checker *NuSMV* [Rob10] is used to generate a simplified canonical representation of the submodels, and the SMT solver *Z3Py*[1] is used for algebraic simplification. Consequently, formal methods are used for correct simplification, which goes beyond their usual purpose of demonstrating correctness. The overall approach includes multiple model-to-model translations including simplifications Δ, equivalence checking E, and a configurable optimization $o \in O$ implemented in *PLCreX* [WS23; WS24b].

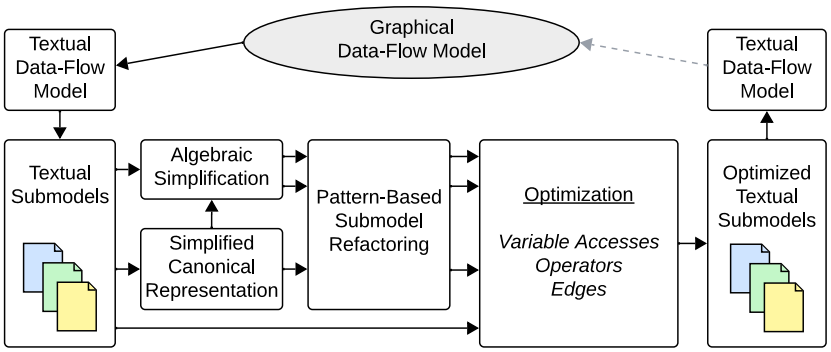


Figure 8.2.: *High-level design flow of the optimization process with focus on the optimization strategy [WS23]*

Challenges:

This approach leads to the following challenges:

1. Identification of potentially optimizable submodels
2. Configurable optimization
3. Correctness of the optimization

8.2. Optimization of Data-Flow Models

This section introduces the optimization of data-flow models, whose low-level design flow is illustrated in Figure 8.3. The transformation $T_{pou \mapsto d}(\omega^{\varphi}_{pou})$ includes the following steps:

1. Δ_1: FBD-to-ST transformation (see Section 8.2.2)
2. Δ_2: Identification of submodels (see Section 8.2.3)

[1] `https://microsoft.github.io/z3guide/`

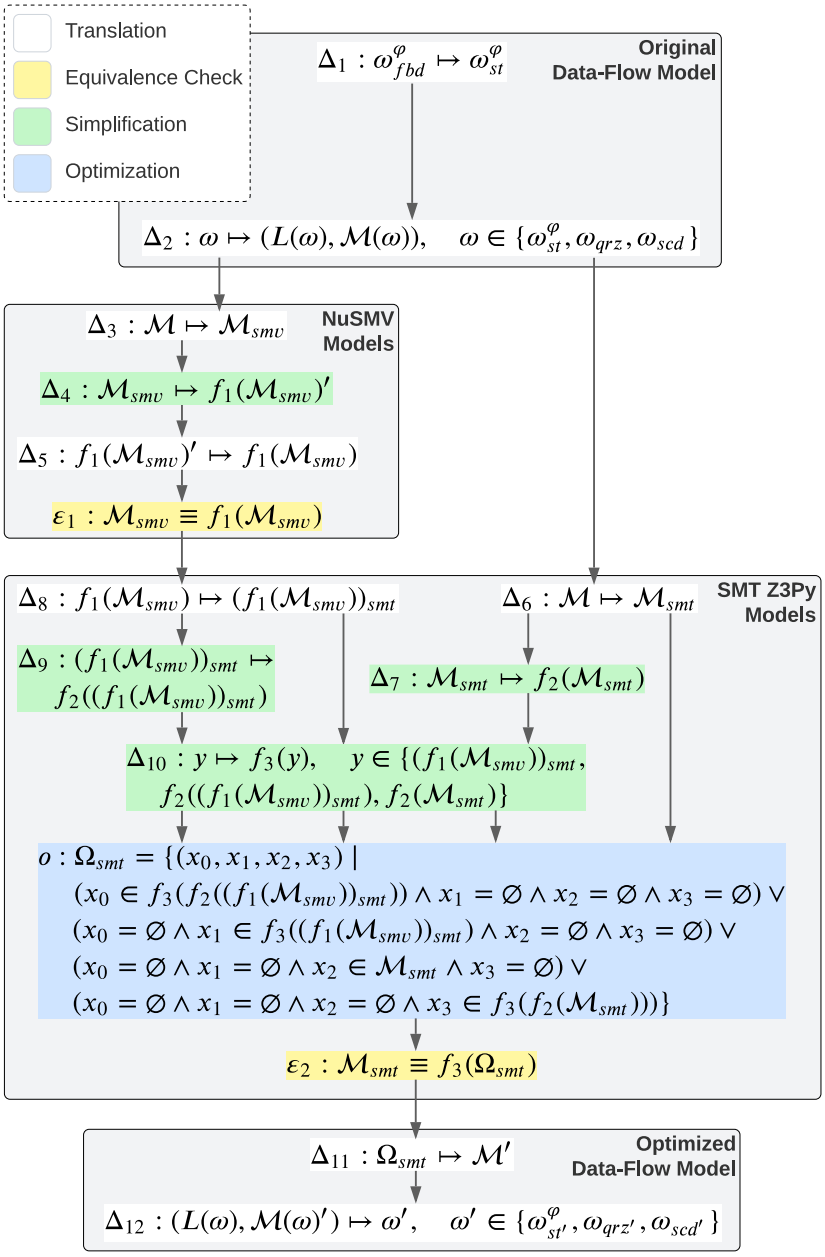


Figure 8.3.: *Low-level design flow of the optimization process [WS23]*

The transformations $T_{qrz \mapsto d}(\omega_{qrz})$ and $T_{scd \mapsto d}(\omega_{scd})$ include the following steps:

1. Δ_2: Identification of submodels (see Section 8.2.3)

The optimization process $T_{d \mapsto d}(\mathcal{M}(\omega_d))$ includes the following steps:

1. Δ_3, Δ_5: *NuSMV*-related translations (see Section 8.2.4 and 8.2.6)
2. Δ_4: *NuSMV*-related simplification (see Section 8.2.5)
3. ϵ_1: Equivalence check using *NuSMV* (see Section 8.2.7)

4. Δ_6, Δ_8: *Z3Py*-related translations (see Section 8.2.8 and 8.2.10)
5. Δ_7, Δ_9: *Z3Py*-related simplification (see Section 8.2.9 and 8.2.11)
6. Δ_{10}: Pattern-based formula refactoring (see Section 8.2.12)
7. o: Configurable optimization (see Section 8.2.13)
8. ϵ_2: Equivalence check using *Z3Py* (see Section 8.2.14)
9. Δ_{11}: Reconstruct submodels (see Section 8.2.15)
10. Δ_{12}: Reconstruct model (see Section 8.2.16)

8.2.1. Operators

Since the optimization and translation process requires multiple model-to-model translations between the different software models, Table 8.1 provides an overview of the operators supported in this approach when translating the original data-flow models ST, SCCharts, and *Quartz* to *NuSMV* and *Z3Py* (and vice versa) [WS23]. This overview is used for several translations explained in the following subsections.

Table 8.1.: *Overview of supported operators across the ST, SCChart,* Quartz, NuSMV, *and* Z3Py *models [WS23]*

ST	SCChart	Quartz	***NuSMV***	***Z3Py***
Boolean operators (with $\alpha(\lambda) = \alpha_{bv}^{bool}$)				
$\tau_{bool}^{not}(\omega_{st}^{\varphi})$	$\tau_{bool}^{not}(\omega_{scd})$	$\tau_{bool}^{not}(\omega_{qrz})$	`!(`λ_1`)`	`Not(`λ_1`)`
$\tau_{bool}^{and}(\omega_{st}^{\varphi})$	$\tau_{bool}^{and}(\omega_{scd})$	$\tau_{bool}^{and}(\omega_{qrz})$	λ_1 `&` ... `&` λ_n	`And(`λ_1`, ...,` λ_n`)`
$\tau_{bool}^{or}(\omega_{st}^{\varphi})$	$\tau_{bool}^{or}(\omega_{scd})$	$\tau_{bool}^{or}(\omega_{qrz})$	λ_1 `|` ... `|` λ_n	`Or(`λ_1 ..., λ_n`)`
$\tau_{bool}^{xor}(\omega_{st}^{\varphi})$	$\tau_{bool}^{xor}(\omega_{scd})$	$\tau_{bool}^{xor}(\omega_{qrz})$	λ_1 `xor` λ_2	`Xor(`λ_1`,` λ_2`)`
Equality/Inequality operators				
$\tau_{comp}^{eq}(\omega_{st}^{\varphi})$	$\tau_{comp}^{eq}(\omega_{scd})$	$\tau_{comp}^{eq}(\omega_{qrz})$	π_1 `=` π_2	π_1 `==` π_2
$\tau_{comp}^{ne}(\omega_{st}^{\varphi})$	$\tau_{comp}^{ne}(\omega_{scd})$	$\tau_{comp}^{ne}(\omega_{qrz})$	π_1 `!=` π_2	π_1 `!=` π_2
Conditional operator				
$\tau_{cond}^{sel}(\omega_{st}^{\varphi})$	$\tau_{cond}^{sel}(\omega_{scd})$	$\tau_{cond}^{sel}(\omega_{qrz})$	λ^b `?` π_1 `:` π_2	`If(`λ^b`,` π_1`,` π_2`)`

Correctness

To check the correctness of the overview, the following lemma is used.

Lemma 8.1. *Let an ST, SCChart or a* Quartz *operator be translated to a corresponding* NuSMV *or* Z3Py *operator (and vice versa) as listed in Table 8.1. Then, the resulting* NuSMV *and* Z3Py *operator conforms to the syntax rules of* NuSMV *and* Z3Py*, and preserves the semantics of the original operator. This is also true after translating the* NuSMV *and* Z3Py *operator back to the original model operator.*

Proof The validity of Lemma 8.1 is proved by a systematic comparison. First, it is checked whether the syntax of the translated operators conforms to the specifications of *NuSMV* [Rob10] and *Z3Py*[2]. Second, the semantic preservation is verified by comparing the specified functionality of the translated *NuSMV* and *Z3Py* operators with that of the original ST, SCChart, and *Quartz* operators, as detailed in Section 3.2, 3.3 and 3.5. Furthermore, it can be concluded that this equivalence holds when operators are translated back from *NuSMV* and *Z3Py* to their respective original model forms, thus ensuring bi-directional semantic consistency.

Illustrative Example for Lemma 8.1

Illustrative examples are implicitly given by the mapping of the operators in Table 8.1.

8.2.2. From Graphical Data-Flow Models to Textual Models

Step Δ_1 of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$ requires the graphical data-flow model to be represented as an equivalent textual model, which is only relevant for $T_{pou\mapsto d}(\omega^{\varphi}_{pou})$, since SCCharts and *Quartz* models are already available in a textual format [WS23]. An FBD-to-ST transformation is often supported by the PLC vendor (like *CODESYS*, *Beremiz*, and others) and also supported by *PLCreX* [WS24b] following the backward translation strategy introduced in Section 6.2.7. In addition, Chapter 6 introduced an FBD-to-*Quartz* transformation and Chapter 7 introduced an FBD-to-SCChart transformation.

8.2.3. Identification of Submodels

Step Δ_2 relevant for $T_{pou\mapsto d}(\omega^{\varphi}_{pou})$, $T_{qrz\mapsto d}(\omega_{qrz})$ and $T_{scd\mapsto d}(\omega_{scd})$ splits the textual data-flow model into potentially optimizable submodels[3] $\mathcal{M}$ and non-modifiable components L [WS23]. The idea is illustrated in Figure 8.4 using an example SCChart. In particular, the model ω_{scd} is split into two sets: (1) set L contains non-modifiable components (like instance `T1`) and (2) set $\mathcal{M}$ contains submodels with potentially optimizable expressions (like m_1).

Completeness

To check the completeness of the strategy, the following lemma is used.

Lemma 8.2. *An FBD, FBD-based ST model, SCChart, and a data-flow oriented* Quartz *model can be completely decomposed into potentially optimizable submodels $\mathcal{M}$ and non-modifiable components L.*

[2] `https://microsoft.github.io/z3guide/`

[3] Arithmetic expressions are treated as submodels, but are not optimized in this approach.

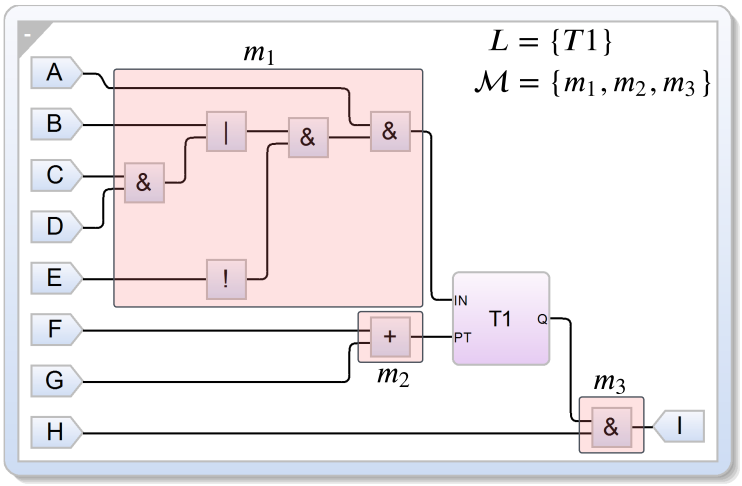


Figure 8.4.: *Visualization of the submodel identification using a simple SCChart example [WS23]*

Proof The validity of Lemma 8.2 is proved by an analysis of the specifications of the original models. More specifically, an FBD, an FBD-based ST model, a data-flow oriented *Quartz* model, and an SCChart are restricted to assignments and model invocations, where *Quartz* models can additionally contain `pause` statements $\Sigma_{pause}(\omega_{qrz})$ (see Section 3.2, 3.3, and 3.5). Thus, a data-flow model can be completely decomposed into the sets L and $\mathcal{M}$, where $\Sigma_{pause}(\omega_{qrz})$ is assigned to L.

Illustrative Example for Lemma 8.2

As an example, the following submodels are identified for the simple SCChart example in Figure 8.4.

```
1  T1 = {A & ( (B | C & D) & !(E)), F+G}
2       ⇒ m1: A & ( (B | C & D) & !(E))
3       ⇒ m2: F + G
4  ------------------------
5  I = T1.Q & H
6       ⇒ m3: T1.Q & H
```

8.2.4. From Submodels $\mathcal{M}$ to SMV Formulas $\mathcal{M}_{smv}$

Step Δ_3 represents the first step of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$, which focuses on a simplification using *NuSMV*. To do this, first each expression $m \in \mathcal{M}$ is translated into a corresponding *NuSMV* formula $m_{smv} \in \mathcal{M}_{smv}$ using a mapping of the operators as listed in Table 8.1 [WS23].

Correctness

To check the correctness of the translation, the following lemma is used.

Lemma 8.3. *A submodel expression m can be translated into a* NuSMV *formula $m_{smv'}$ if m contains operators that are restricted to those listed in Table 8.1. $m_{smv'}$ conforms to the syntax rules of m_{smv} and preserves the semantics of m.*

Proof The validity of Lemma 8.3 follows from Lemma 8.1.

Illustrative Example for Lemma 8.3

As an example, m_1 of the simple SCChart example in Figure 8.4 is translated to $m_{1,smv}$ as follows. According to Table 8.1, the operators are syntactically equivalent. Therefore, no changes result.

```
1    A & ( (B | C & D) & !(E))
2        ⇒ (A&((B|(C&D))&!E))
```

8.2.5. f_1-Simplification of $\mathcal{M}_{smv}$

Once $\mathcal{M}_{smv}$ has been created, in step Δ_4 of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$, each formula $m_{smv} \in \mathcal{M}_{smv}$ will be expressed as an equivalent simplified formula $f_1(m_{smv})' \in f_1(\mathcal{M}_{smv})'$ using a built-in simplification of *NuSMV*[4]. Depending on the underlying complexity of the formula, this may lead to an initial simplification of the submodel [WS23]. In this context, it may happen that besides the operators listed in Table 8.1, additional temporary variables and `case` expressions appear (which still follow the syntactical rules) [Rob10].

8.2.6. From $f_1(\mathcal{M}_{smv})'$ to SMV Formulas $f_1(\mathcal{M}_{smv})$

Due to the possibility of additional `case` expressions and additional temporary variables in $f_1(\mathcal{M}_{smv})'$, in step Δ_5 of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$, a translation back to the expressions restricted to the operators listed in Table 8.1 is required and realized by a model-to-model transformation based on the *NuSMV* grammar [Rob10]. Additionally, a reduction to the set of original variables is required [WS23].

8.2.7. Equivalence Check of $\mathcal{M}_{smv}$ and $f_1(\mathcal{M}_{smv})$

Step ϵ_1 of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$ ensures the correctness of $\mathcal{M}_{smv}$ and $f_1(\mathcal{M}_{smv})$ by an equivalence check using *NuSMV*.

Correctness

To check the correctness of the simplification and back translation, the following lemma is used.

[4] Simplification is triggered by printing the formula.

Lemma 8.4. *A* NuSMV *formula m_{smv} can be expressed as an equivalent simplified formula $f_1(m_{smv})'$, which possibly contains additional temporary variables and* `case` *expressions. After translating $f_1(m_{smv})'$ back into an expression restricted to the operators listed in Table 8.1 following the syntax rules of* NuSMV*, the result is a possibly simplified formula $f_1(m_{smv'})$ that conforms to the syntax rules of $f_1(m_{smv})$ and preserves the semantics of m_{smv}.*

Proof The validity of Lemma 8.4, i.e., step Δ_4 and Δ_5, is proved for each formula during runtime by an equivalence check ϵ_1 using *NuSMV* as illustrated in the example below. In particular, *NuSMV* is used to check whether for all elements $m \in \mathcal{M}$ the following assertion holds: $m_{smv} \equiv f_1(m_{smv})$.

Illustrative Example for Lemma 8.4

As an illustrative example, the following snippet illustrates the equivalence check using the derived submodel m_2 of the `FBD_POLL` example:

```
m_{2,smv}: ((!(IN3)&IN2&IN1)|(IN3&!(IN2)&IN1)|(IN3&IN2&!(IN1)))
    // Δ4:
    ⇒ FORMULA =
    ⇒ case
    ⇒ IN1 : case
    ⇒ IN2 : !IN3;
    ⇒ TRUE : IN3;
    ⇒ esac;
    ⇒ TRUE : (IN2 & IN3);
    ⇒ esac
    -----------------------
    // Δ5:
    ⇒ f1(m_{2,smv}): (IN1?(IN2?!IN3:IN3):(IN2&IN3))
    -----------------------
    // ε1 : m_{2,smv} = f1(m_{2,smv}):
    ⇒ ((!(IN3)&IN2&IN1)|(IN3&!(IN2)&IN1)|(IN3&IN2&!(IN1)))=(IN1?(!IN2&!
        IN3):(IN1?(IN2?!IN3:IN3):(IN2&IN3))
    ⇒ FORMULA = TRUE
```

8.2.8. From Submodels $\mathcal{M}$ to SMT Formulas $\mathcal{M}_{smt}$

Similar to the translation of the submodels from $\mathcal{M}$ to $\mathcal{M}_{smv}$, in step Δ_6 of the optimization process $T_{d \mapsto d}(\mathcal{M}(\omega_d))$, each expression of the submodels $m \in \mathcal{M}$ is translated into a corresponding *Z3Py* formula $m_{smt} \in \mathcal{M}_{smt}$ using a mapping of operators as listed in Table 8.1 [WS23].

Correctness

To check the correctness of the translation, the following lemma is used.

Lemma 8.5. *A submodel expression m can be translated into a* Z3Py *formula $m_{smt'}$ if m contains operators that are restricted to those listed in Table 8.1. $m_{smt'}$ conforms to the syntax rules of m_{smt} and preserves the semantics of m.*

Proof The validity of Lemma 8.5 follows from Lemma 8.1.

Illustrative Example for Lemma 8.5

As an example, m_1 of the simple SCChart example in Figure 8.4 is translated to $m_{1,smt}$ as follows.

```
1     A & ( (B | C & D) & !(E))
2         ⇒ And(A,And(Or(B,And(C,D)),Not(E)))
```

8.2.9. f_2-Simplification of $\mathcal{M}_{smt}$

Once $\mathcal{M}_{smt}$ has been created, in step Δ_7 of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$, each formula $m_{smt} \in \mathcal{M}_{smt}$ will be expressed as an equivalent simplified formula $f_2(m_{smt})' \in f_2(\mathcal{M}_{smt})'$ using the built-in simplification feature of *Z3Py*, such as eliminating unnecessary terms, using associative and commutative properties, and solving equations [WS23]. Although there are no *Z3Py*-based methods to ensure that expressions are simplified to a unique canonical form[5], simplification by *Z3Py* is considered as a possible subsequent simplification of submodels already simplified by *NuSMV* $f_1(\mathcal{M}_{smv})$.

8.2.10. From $f_1(\mathcal{M}_{smv})$ to SMT Formulas $(f_1(\mathcal{M}_{smv}))_{smt}$

In step Δ_8 of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$, $f_1(\mathcal{M}_{smv})$ is translated into equivalent *Z3Py* formulas $(f_1(\mathcal{M}_{smv}))_{smt}$ for a subsequent simplification (depending on the optimization configuration and formula complexity) [WS23]. This translation is restricted to the operators listed in Table 8.1, which is implicitly given by the fact that $f_1(\mathcal{M}_{smv})$ is already restricted to these operators.

8.2.11. f_2-Simplification of $(f_1(\mathcal{M}_{smv}))_{smt}$

After the simplified *NuSMV* formulas $f_1(\mathcal{M}_{smv})$ have been translated into corresponding *Z3Py* formulas $(f_1(\mathcal{M}_{smv}))_{smt}$, a further simplification is performed using *Z3Py*'s built-in simplification feature in step Δ_9 of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$, as introduced in Section 8.2.9, denoted as $f_2((f_1(\mathcal{M}_{smv}))_{smt})$ [WS23].

[5] `https://microsoft.github.io/z3guide/`

8.2.12. Pattern-Based Formula Refactoring

Comparing each submodel $m \in \mathcal{M}$ with the corresponding formula in $\mathcal{M}_{smt}$, $f_2(\mathcal{M}_{smt})$, $(f_1(\mathcal{M}_{smv}))_{smt}$, and $f_2((f_1(\mathcal{M}_{smv}))_{smt})$, it can be concluded that, depending on the complexity of the original submodel, different simplification approaches can have different effects on the following metrics:

- ***Number of Operators* (NoO):** Number of visible operators (without instances) in the corresponding graphical data-flow model
- ***Number of Edges* (NoE):** Number of visible edges in the corresponding graphical data-flow model
- ***Number of Variable Accesses* (NoV):** Sum of visible input variables and connected instance output variables, i.e., variables accessed by the submodels

Overall, there is no guarantee that the corresponding formula represents the most optimal solution. Thus, as an additional optimization approach, step Δ_{10} of the optimization process $T_{d \mapsto d}(\mathcal{M}(\omega_d))$ considers a pattern-based formula refactoring with a focus on exclusive or patterns.

Definition 8.1 (Exclusive Or Pattern). *An* exclusive or *operator expressed with the operators listed in Table 8.1 can have one of the following patterns within* Z3Py*:*

- ***Pattern 1: If-Not***

```
1 |   If(τ1,Not(τ2),τ2);
```

- ***Pattern 2: Not-Equal***

```
1 |   Not(Eq(τ1,τ2));
```

In both scenarios, the pattern can be replaced through `Xor(`τ_1`,`τ_2`)`*.*

Correctness

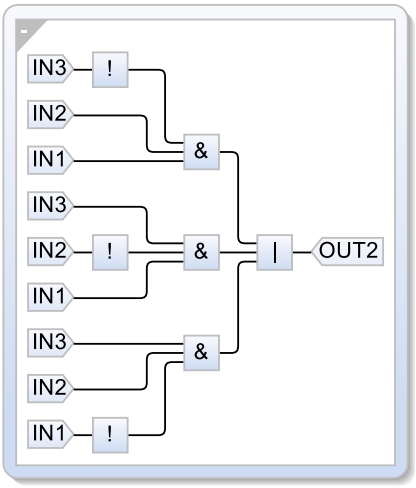
To check the correctness of Definition 8.1, the following lemma is used.

Lemma 8.6. *Both patterns listed in Definition 8.1 conform to the logic of* `Xor(`τ_1`,`τ_2`)` *and thus, can be expressed as* `Xor(`τ_1`,`τ_2`)`*, which reduces the number of variable accesses compared to pattern 1 and reduces the number of operators compared to patterns 1 and 2.*

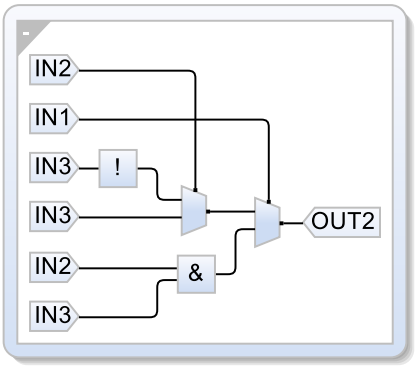
Proof The validity of Lemma 8.6 is proved by an equivalence check of the exclusive or operator and both patterns using *Z3* theorem proving. In particular, *Z3Py* is used to check whether for any assignment of the variables τ_1 and τ_2 the following assertion holds: `pattern 1` $\equiv$ `Xor(`τ_1`,`τ_2`)` and `pattern 2` $\equiv$ `Xor(`τ_1`,`τ_2`)`. In addition, the reduction to one operator is implicitly confirmed by the refactoring strategy. In pattern 1, the variable access is reduced to two times.

Illustrative Example for Lemma 8.6

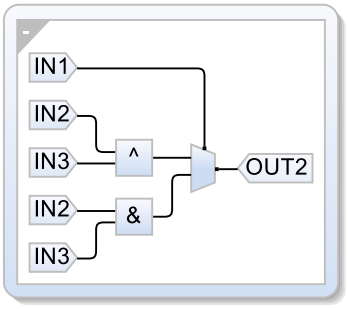
As an example, Figure 8.5 illustrates pattern 1, which is identified in $\omega_{2,smt}$ of the FBD_POLL example. The original submodel is shown in Figure 8.5a and the optimization is shown in Figure 8.5b. The pattern-based formula refactoring reduces NoO, NoE, and NoV, as shown in Figure 8.5c.



(a) *Original submodel m_2: NoO = 7, NoE = 16, NoV = 9*



(b) *Simplified model (op-opt): NoO = 4, NoE = 10, NoV = 6*



(c) *Refactored model: NoO = 3, NoE = 8, NoV = 5*

Figure 8.5.: *Simplification of m_2 of the FBD_POLL example (with and without pattern-based formula refactoring)*

8.2.13. Selection of Optimized SMT Formulas

With the goal of optimizing the original software model, the question arises which corresponding element of the four sets should be used for each submodel to create the set Ω_{smt}. To do this, in step o of the optimization process $T_{d \mapsto d}(\mathcal{M}(\omega_d))$, the resulting formulas can be rated using the metrics NoO, NoE, and NoV. While these metrics provide a possible indication of the complexity of the resulting submodels, the perceived complexity depends on the preferences of the individual user. Consequently, there is a risk that

with a fixed choice of one of the corresponding formula in $\mathcal{M}_{smt}$, $f_2(\mathcal{M}_{smt})$, $(f_1(\mathcal{M}_{smv}))_{smt}$, and $f_2((f_1(\mathcal{M}_{smv}))_{smt})$, the resulting optimization with respect to NoO, NoE, and NoV may not fully satisfy the user's preferences [WS23].

The problem is illustrated by two simplifications in Figure 8.6 applied to m_3 of the `Cylinder_Control_System` example. NoO is higher in Scenario 1 in Figure 8.6a, but leads to a lower NoE and NoV compared to Scenario 2 in Figure 8.6b. Thus, it can be concluded that not all metrics can be optimized equally following a single simplification strategy, which is why the user can choose [WS23]. This gives the user the flexibility to manually select the optimization strategy based on personal preference. Possible optimization strategies according to the introduced metrics are *variable access* (`var-opt`), *number of operators* (`op-opt`), and *number of edges* (`edge-opt`).

More specifically, the set of optimized formulas Ω_{smt} is created as follows, where exactly one element from one of the four introduced sets is selected as the resulting optimization of a formula $\omega_{smt} \in \Omega_{smt}$ (depending on the chosen optimization strategy and the metrics of each corresponding formula in the different sets, as demonstrated in Section 8.3):

$$
\begin{aligned}
\Omega_{smt} = \{(x_0, x_1, x_2, x_3) \mid \\
&(x_0 \in f_3(f_2((f_1(\mathcal{M}_{smv}))_{smt})) \wedge x_1 = \varnothing \wedge x_2 = \varnothing \wedge x_3 = \varnothing) \vee \\
&(x_0 = \varnothing \wedge x_1 \in f_3((f_1(\mathcal{M}_{smv}))_{smt}) \wedge x_2 = \varnothing \wedge x_3 = \varnothing) \vee \\
&(x_0 = \varnothing \wedge x_1 = \varnothing \wedge x_2 \in \mathcal{M}_{smt} \wedge x_3 = \varnothing) \vee \\
&(x_0 = \varnothing \wedge x_1 = \varnothing \wedge x_2 = \varnothing \wedge x_3 \in f_3(f_2(\mathcal{M}_{smt})))\}
\end{aligned}
$$

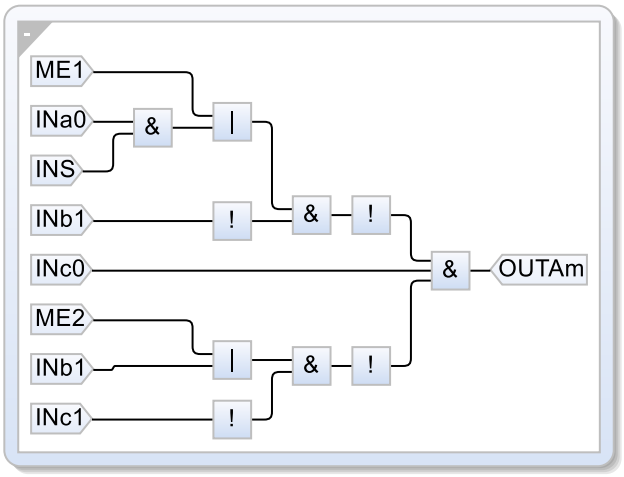
8.2.14. Equivalence Check of $\mathcal{M}_{smt}$ and Ω_{smt}

As a previous step before the optimized submodel is reconstructed, in step ϵ_2 of the optimization process $T_{d \mapsto d}(\mathcal{M}(\omega_d))$ it is checked whether the optimized formulas Ω_{smt} are equivalent to $\mathcal{M}_{smt}$. Thus, this steps checks the correctness of the translation in step Δ_8, simplifications in step Δ_7, Δ_9 and Δ_{10}, and optimization in step o.

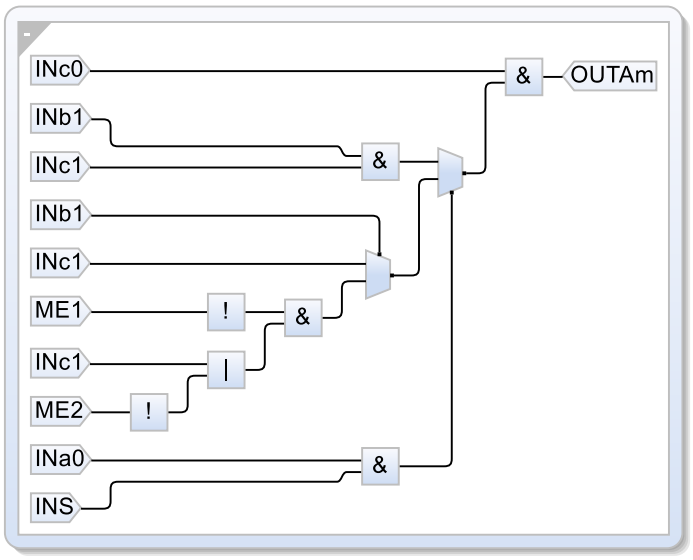
Correctness

To check the correctness of the translation, simplifications and optimization, the following lemma is used.

Lemma 8.7. *Each optimized formula $\omega_{smt'} \in \Omega_{smt}$ conforms to the syntax rules of ω_{smt} and preserves the semantics of $m_{smt} \in \mathcal{M}_{smt}$, which implicitly confirms the correctness of translation Δ_8, simplifications Δ_7, Δ_9, and Δ_{10}, and optimization o.*



(a) *Simplification Scenario 1: NoO = 10, NoE = 18, NoV = 8*



(b) *Simplification Scenario 2: NoO = 9, NoE = 19, NoV = 10*

Figure 8.6.: *Comparison of two different simplification scenarios related to m_3 of the `Cylinder_Control_System` example*

Proof The validity of Lemma 8.7 is proved by an equivalence check for each formula during runtime using *Z3Py*. In particular, similar to Lemma 8.6, it is checked whether for all elements $m_{smt} \in \mathcal{M}_{smt}$ the following assertion holds: $\omega_{smt'} \equiv m_{smt}$. This implicitly confirms the syntactic correctness of $\omega_{smt'}$.

Illustrative Example for Lemma 8.7

As an illustrative example, the following snippet illustrates the equivalence check using m_2 of the `FBD_POLL` example and `op-opt` optimization strategy:

```
// Δ6 :
m2,smt : Or(And(IN1, IN2, Not(IN3)), And( IN1, Not(IN2), IN3), And(Not(IN1
    ), IN2, IN3))
```

```
3  ----------------------
4  // Δ7:
5  f2(m2,smt): Or(And(IN1, IN2, Not(IN3)), And(IN1, Not(IN2), IN3), And(Not(
       IN1), IN2, IN3))
6  ----------------------
7  // Δ8:
8  (f1(m2,smv))smt: If(IN1,If(IN2,Not(IN3),IN3),(And(IN2,IN3)))
9  ----------------------
10 // Δ9:
11 f2((f1(m2,smv))smt): If(IN1, Not(IN3 == IN2), And(IN2, IN3))
12 ----------------------
13 // Δ7 ⇒ Δ10:
14 no pattern identified
15 ----------------------
16 // Δ8 ⇒ Δ10:
17 Pattern 1 identified
18     ⇒ If(IN1,Xor(IN2,IN3),And(IN2,IN3))
19 ----------------------
20 // Δ9 ⇒ Δ10:
21 Pattern 2 identified
22     ⇒ If(IN1,Xor(IN3,IN2),And(IN2,IN3))
23 ----------------------
24 // o: selected formula: Δ8 ⇒ Δ10
25 op-opt: If(IN1,Xor(IN2,IN3),And(IN2,IN3))
26 ----------------------
27 // ϵ2:
28 solve((Or(And(IN1, IN2, Not(IN3)), And( IN1, Not(IN2), IN3), And(Not(IN1
       ), IN2, IN3))!=If(IN1,Xor(IN2,IN3),And(IN2,IN3))))
29     ⇒ no solution
```

8.2.15. From Ω_{smt} to Initial Submodels $\mathcal{M}'$

If the correctness is confirmed by the equivalence check in step ϵ_2, the step Δ_{11} of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$ translates Ω_{smt} back to the submodels $\mathcal{M}'$. Since the set Ω_{smt} is available as *Z3Py* formulas and thus restricted to the operators listed in Table 8.1, the translation back to $\mathcal{M}'$ is straightforward (see Section 8.2.1) [WS23].

8.2.16. Reconstruct Software Model

In step Δ_{12} of the optimization process $T_{d\mapsto d}(\mathcal{M}(\omega_d))$, the original submodels $\mathcal{M}$ are replaced through the optimized and back translated submodels $\mathcal{M}'$. Combined with the non-modifiable components, this leads to the optimized software model ω'.

Correctness

To check the correctness of the reconstruction, the following lemma is used.

Lemma 8.8. *The resulting optimized software model ω' conforms to the syntax rules of ω and preserves the semantics of the original software model, which implicitly confirms the correctness of the back translations Δ_{11} and Δ_{12}.*

Proof The validity of Lemma 8.8 is proved by the correctness of the optimized formulas (see Lemma 8.7) and by restrictions to the supported operators listed in Table 8.1, whose correctness follows from Lemma 8.1.

Illustrative Example for Lemma 8.8

As an example, the following model represents an optimized model of the simple SCChart example in Figure 8.4.

```
1    T1 = {A & ( (B | C & D) & !(E)), F+G}
2        ⇒ T1 = {A & (!(E)) & (C?(D|B):B), F+G}
3    ------------------------
4    I = T1.Q & H
5        ⇒ T1.Q & H
```

8.3. Experimental Results

Following the optimization approach presented in the previous section, this section identifies the optimization potential for data-flow models using industrial examples from PLC vendors and literature. The optimization potential is evaluated using the system architecture shown in the high-level design flow in Figure 8.1 in Section 8.1. For a fair evaluation of the optimization possibilities, Table 8.2 lists a collection of representative data-flow models with the number of potentially optimizable submodels and the average runtime for determining all four optimized submodels with code generation for the selected strategy[6]. These models have been manually implemented as data-flow oriented SCCharts in *KIELER* and are optimized using the optimization process implemented as a prototype in *PLCreX*[7]. The resulting optimized SCCharts are compiled in *KIELER* to check their syntactic correctness. The selection of real-world examples from PLC vendors and literature ensures practical relevance and objectivity when evaluating the optimization potential using the introduced optimization process. In total, 96 submodels are available as input models, which results in 288 optimized output models after optimization.

The optimized submodels are listed in detail in Appendix L in a textual ST-like format[8], which allows to reproduce the experimental results in Table 8.3 that summarizes the experimental results of the case study. Additionally, two results are shown in Figure 8.7 to demonstrate that the optimization strategy can affect the other metrics. For example, while the `op-opt` strategy reduces the number of operators in Figure 8.7a, it increases the number of variable accesses. Similarly, optimizing the model in Figure 8.7b with respect to `op-opt` saves less number of variable accesses compared to the other strategies. Thus, of particular interest is the comparison of the average optimization potential using the strategies `op-opt`, `edge-opt`, and `var-opt`, evaluated by the NoE, NoO, and NoV metrics for the resulting model. The results are summarized

[6] The case study was tested on Windows 11, x64, 11th Gen Intel(R) Core(TM) i7-11850H, 2.50GHz, 8 cores, 32GB RAM DDR4, with Python 3.9.18. The average refers to the mean of the three optimization strategies.

[7] The latest release supports FBDs as input models.

[8] This format represents an intermediate *PLCreX* format.

Table 8.2.: *Data-flow model overview, including number of submodels, number of operators (without instances), and average runtime for determining all four optimized submodels with code generation for the selected strategy (based on related work without pattern-based formula refactoring [WS23])*

ID	Data-Flow Model	Number of Submodels	Number of Operators	Ø Runtime	Source
1	Air Condition Control	4	9	2.04 s	[Tap15]
2	Antivalence 3x	1	10	1.15 s	[Kar18]
3	Bending Machine Control	15	21	5.21 s	[AG20]
4	Cylinder Control System	6	30	3.59 s	[Sch14]
5	Dice Numbers Indicator	9	67	6.58 s	[Kar18]
6	KV Diagram optimized Chart	2	13	1.51 s	[Kar18]
7	Pollutant Indicator	3	18	2.07 s	[Bub17]
8	Reservoirs Control System 1	4	35	3.34 s	[WZ07]
9	Reservoirs Control System 2	4	35	3.31 s	[Kar18]
10	Roll Down Shutters	2	24	2.14 s	[AG20]
11	Cable winch	4	28	2.69 s	[Tap15]
12	Seven Segment Display	7	81	5.93 s	[WZ07]
13	Shop Window Lighting	8	15	3.11 s	[AG20]
14	Silo Valve Control System	1	11	1.22 s	[WZ07]
15	Smoke Detection System	3	28	2.32 s	[Tap15]
16	Sports Hall Lighting	12	17	4.29 s	[AG20]
17	Thermometer Code System	3	10	1.76 s	[Bub17]
18	Toggle Switch 4x	1	27	1.88 s	[Bub17]
19	Ventilation Control System	3	25	2.42 s	[Kar18]
20	Wind Direction Indicator	4	34	2.78 s	[Bub17]

in Table 8.4. As expected, each optimization strategy leads to the best improvement of the corresponding metric. In detail, the optimization approach and examples show an optimization potential of about 35 % for NoO with the `op-opt` strategy (on average), about 30 % for NoE with the `edge-opt` strategy (on average), and about 26 % for NoV with the `var-opt` strategy (on average). The `edge-opt` strategy leads to the same optimization of the NoV metric. Overall, as in the case study without pattern-based formula refactoring [WS23], it can be concluded that the `edge-opt` strategy is the best choice in terms of the NoE, NoO, and NoV metrics, since it tends to produce the best balanced optimizations, even if NoO is not reduced as much as with the `op-opt` strategy (on average).

Table 8.3.: *Experimental results of the case study with values in percent relative to the non-optimized model ranging from -75% (better) to 10.7% (worse) (based on experimental results of related work without pattern-based formula refactoring [WS23])*

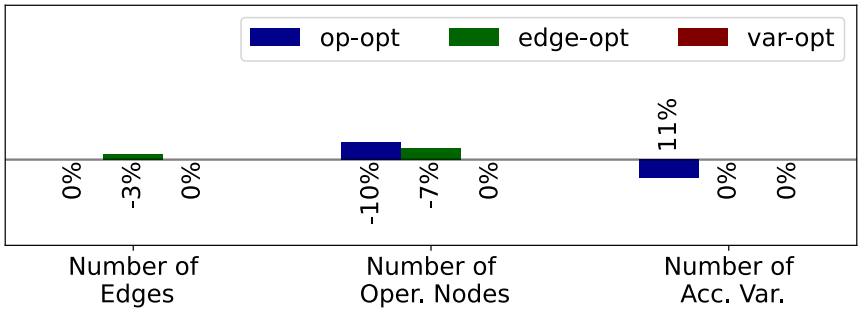
	op-opt			edge-opt			var-opt		
ID	**NoE**	**NoO**	**NoV**	**NoE**	**NoO**	**NoV**	**NoE**	**NoO**	**NoV**
1	-5.0	-11.1	0.0	-5.0	-11.1	0.0	0.0	0.0	0.0
2	-47.4	-50.0	-44.4	-47.4	-50.0	-44.4	-47.4	-50.0	-44.4
3	-15.7	-28.6	-6.7	-15.7	-28.6	-6.7	-9.8	-14.3	-6.7
4	0.0	-10.0	10.7	-3.4	-6.7	0.0	0.0	0.0	0.0
5	-9.8	-14.9	-4.5	-9.8	-11.9	-7.6	-9.8	-11.9	-7.6
6	-28.0	-38.5	-16.7	-28.0	-38.5	-16.7	-28.0	-38.5	-16.7
7	-43.6	-50.0	-38.1	-43.6	-50.0	-38.1	-43.6	-50.0	-38.1
8	-3.1	-2.9	-3.3	-4.6	0.0	-10.0	-4.6	0.0	-10.0
9	-13.6	-8.6	-19.4	-13.6	-8.6	-19.4	-13.6	-8.6	-19.4
10	-21.7	-29.2	-13.6	-21.7	-29.2	-13.6	-21.7	-29.2	-13.6
11	-44.0	-57.1	-27.3	-44.0	-57.1	-27.3	-28.0	-28.6	-27.3
12	-31.3	-34.6	-27.8	-31.3	-34.6	-27.8	-28.8	-29.6	-27.8
13	-16.2	-26.7	-9.1	-16.2	-26.7	-9.1	-16.2	-26.7	-9.1
14	-65.2	-72.7	-58.3	-65.2	-72.7	-58.3	-65.2	-72.7	-58.3
15	-48.0	-50.0	-45.0	-48.0	-50.0	-45.0	-48.0	-50.0	-45.0
16	-7.1	-17.6	0.0	-7.1	-17.6	0.0	-7.1	-17.6	0.0
17	-42.9	-50.0	-38.9	-42.9	-50.0	-38.9	-42.9	-50.0	-38.9
18	-72.9	-70.4	-75.0	-72.9	-70.4	-75.0	-72.9	-70.4	-75.0
19	-23.2	-24.0	-22.6	-28.6	-24.0	-32.3	-28.6	-24.0	-32.3
20	-62.9	-58.8	-66.7	-62.9	-58.8	-66.7	-62.9	-58.8	-66.7

Table 8.4.: *Average number of edges, operators, and variable accesses after optimization*

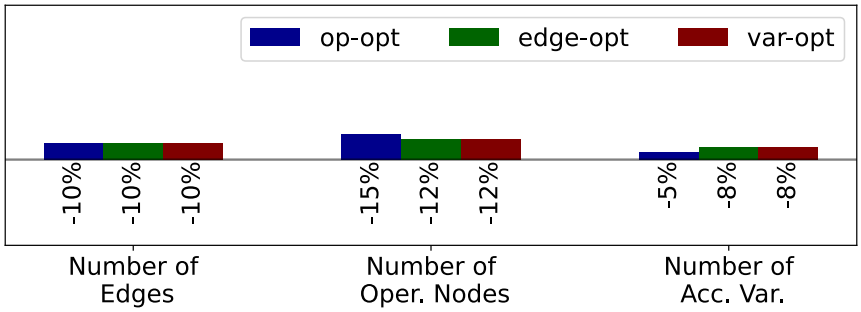
Optimization Strategy	Ø Number of Edges	Ø Number of Operators	Ø Number of Var. Accesses
op-opt	-30.07 %	-35.28 %	-25.33 %
edge-opt	-30.59 %	-34.82 %	-26.83 %
var-opt	-28.95 %	-31.54 %	-26.83 %

8.4. Summary

This chapter introduced a formal methods-based optimization of data-flow models using *NuSMV*, SMT *Z3Py*, and a pattern-based refactoring approach. For this purpose, several model-to-model transformations, simplifications, equivalence checks, and an optimization were defined, with which the optimization potential of a set of real-world data-flow models has been identified. Based on the presented lemmas and experimental results, the following theorem encapsulates the entire optimization:



(a) *Cylinder Control System*



(b) *Dice Numbers Indicator*

Figure 8.7.: *Experimental Results*

Theorem 8.1 (Formal Methods-Based Optimization of Data-Flow Models). *Let* $\omega \in \{\omega_{st}^{\vartheta}, \omega_{qrz}, \omega_{scd}\}$ *be a textual data-flow model that is translated according to the transformations* $T_{pou \mapsto d}(\omega_{pou}^{\varphi})$, $T_{qrz \mapsto d}(\omega_{qrz})$ *and* $T_{scd \mapsto d}(\omega_{scd})$, *and optimized according to the optimization process* $T_{d \mapsto d}(\mathcal{M}(\omega_d))$ *introduced in this chapter. Then, the resulting model* ω'*:*

1. *Can be completely decomposed into potentially optimizable submodels and non-modifiable submodels*
2. *Is optimized for one of the following metrics:*
   - *Number of operators (NoO)*
   - *Number of edges (NoE)*
   - *Number of variable accesses (NoV)*
3. *Conforms to the syntax rules of* ω *and preserves the semantics of* ω

Proof The validity of Theorem 8.1 is proved as follows:

1. **Submodels:** Lemma 8.2 shows that a data-flow model can be completely decomposed into the sets of non-modifiable submodels L and

potentially optimizable submodels $\mathcal{M}$ (see Section 8.2.3).

2. **Optimization Potential:** Experimental results on real-world models show an optimization potential with respect to the NoO metric of about 35.2 % on average, with respect to the NoE metric of about 30.5 % on average, and with respect to the NoV metric of about 26.8 % on average (see Section 8.3).
3. **Syntax Conformance and Semantic Preservation:** According to Lemma 8.1, the operators are restricted to those listed in Table 8.1, which are supported by all of the considered software models (see Section 3.5 and 3.2). Furthermore, the following lemmas and theorems addresses the preservation of semantics during the optimization process, which are structured as follows:
   - Δ_2**:** Lemma 8.2
   - Δ_3**:** Lemma 8.3
   - Δ_4**,** Δ_5**:** Lemma 8.4
   - Δ_6**:** Lemma 8.5
   - Δ_{10}**:** Lemma 8.6
   - Δ_7**,** Δ_8**,** Δ_9**,** Δ_{10}**,** o**:** Lemma 8.7
   - Δ_{11}**,** Δ_{12}**:** Lemma 8.8

Overall, this results in the following solutions to the challenges summarized in Section 8.1.

1. **Identification of potentially optimizable submodels:** The solution follows from Section 8.2.3.
2. **Configurable optimization:** The solution follows from Section 8.2.13.
3. **Correctness of the optimization:** The solution follows from Section 8.2.

Chapter 9

Control-Flow Oriented SCCharts of POU-Based *Quartz* Models

Contents

This chapter focuses on a code refactoring of FBD-based and ST-based *Quartz* models and their transformation to control-flow oriented SCCharts, providing

an alternative view for system analysis. The goal is to create a robust set of translation functions that ensure semantic preservation during the transition. More specifically, this chapter details a pattern-based code refactoring of *Quartz* models and a set of detailed translation functions. In addition to the approaches presented in [WS22], it considers the following additional issues:

- **Model Declaration:** Translation of model declaration
- **Interfaces and Variables:** Translation of interfaces and variables
- **Data Types and Fields:** Translation of data types and fields
- **Expressions:** Translation of expressions
- **Statements:** Additional *Quartz* statements to extend the approach from FBD-based *Quartz* models to ST-based *Quartz* models

The correctness of the translation functions is proved by theoretical reasoning, which includes a detailed analysis of the resulting syntax and semantics compared to the syntax rules and semantics specified in Chapter 3. In addition, the theoretical results are evaluated with real-world and self-defined *Quartz* models.

The outline of this chapter is as follows: Section 9.1 introduces the high-level design flow and translation strategy. Section 9.2 defines a pattern-based code refactoring of *Quartz* models, Section 9.3 defines the translation functions and theoretical analysis, and Section 9.4 defines possible SCChart optimizations. Section 9.5 presents an evaluation of the theoretical results, and Section 9.6 summarizes the transformation.

9.1. High-Level Design Flow – Quartz-to-SCChart

The high-level design flow for transforming an FBD-based or ST-based *Quartz* model $\omega_{qrz} \in \Omega_{qrz}$ to a control-flow oriented SCChart $\omega_{scc} \in \Omega_{scc}$ is shown in Figure 9.1.

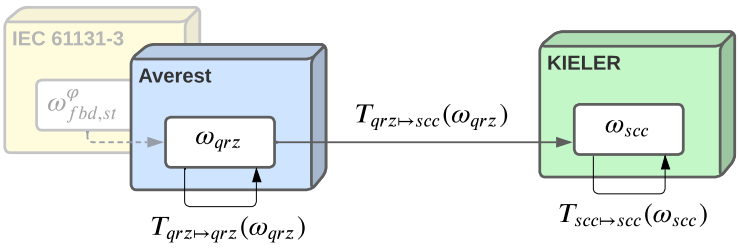


Figure 9.1.: *High-level design flow of the Quartz-to-SCChart transformation*

The *Quartz* code refactoring $T_{qrz\mapsto qrz}(\omega_{qrz})$ includes the following refactoring approach:

- $t^{\Sigma_{P1}}_{qrz\mapsto qrz}(\Sigma_{P1}(\omega_{qrz}))$: Pattern-based *Quartz* code refactoring (see Section 9.2)

In addition, the Quartz-to-SCChart transformation $T_{qrz\mapsto scc}(\omega_{qrz})$ includes the following transformation steps:

1. $t^{\delta_\omega}_{qrz\mapsto scc}(\delta_\omega(\omega_{qrz}))$: Model declaration (see Section 9.3.1)
2. $t^{\Delta_{idcl}}_{qrz\mapsto scc}(\Delta_{idcl}(\omega_{qrz}))$: Interfaces (see Section 9.3.2)
3. $t^{\Delta_{vdcl}}_{qrz\mapsto scc}(\Delta_{local}(\omega_{qrz}))$: Variables (see Section 9.3.3)
4. $t^{\alpha}_{qrz\mapsto scc}(\alpha^{[+]}(\omega_{qrz}))$: Data types and fields (see Section 9.3.4)
5. $t^{\tau}_{qrz\mapsto scc}(\tau(\omega_{qrz}))$: Expressions (see Section 9.3.5)
6. $t^{\Sigma_{nothing}}_{qrz\mapsto scc}(p_{nothing}(\omega_{qrz}))$: Immediate Transition (see Section 9.3.6)
7. $t^{\Sigma_{await}}_{qrz\mapsto scc}(p^{\vartheta}_{await}(\omega_{qrz}))$: Await (see Section 9.3.7)
8. $t^{\Sigma_{pause}}_{qrz\mapsto scc}(p_{pause}(\omega_{qrz}))$: Pause (see Section 9.3.8)
9. $t^{\Sigma_{ass}}_{qrz\mapsto scc}(p^{\vartheta}_{ass}(\omega_{qrz}))$: Assignments (see Section 9.3.9)
10. $t^{\Sigma_{conc}}_{qrz\mapsto scc}(\sigma_{conc}(\omega_{qrz}))$: Synchronous concurrency (see Section 9.3.10)
11. $t^{\Sigma_{loop}}_{qrz\mapsto scc}(p^{\vartheta}_{loop}(\omega_{qrz}))$: Loops (see Section 9.3.11)
12. $t^{\Sigma_{halt}}_{qrz\mapsto scc}(\sigma_{halt}(\omega_{qrz}))$: Halt (see Section 9.3.12)
13. $t^{\Sigma_{abort}}_{qrz\mapsto scc}(p^{\vartheta}_{abort}(\omega_{qrz}))$: Abort (see Section 9.3.13)
14. $t^{\Sigma_{cond}}_{qrz\mapsto scc}(p^{\vartheta}_{cond}(\omega_{qrz}))$: Conditions (see Section 9.3.14)
15. $t^{\Sigma_{seq}}_{qrz\mapsto scc}(\Sigma_{seq}(\omega_{scc'}))$: Sequences (see Section 9.3.15)

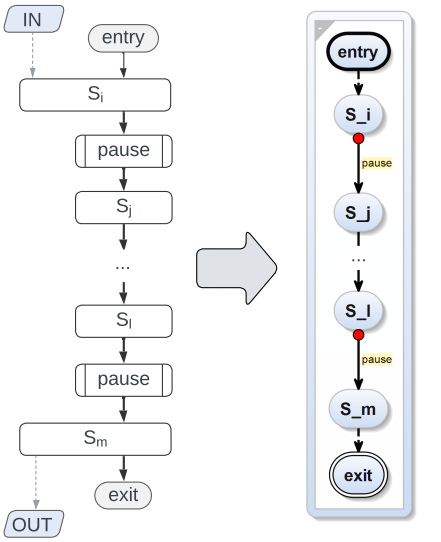
Furthermore, the transformation includes the following SCChart optimizations $T_{scc\mapsto scc}(\omega_{scc})$:

- $t^{\Sigma_{hierarchy}}_{scc\mapsto scc}(\Sigma(\omega_{scc}))$: Flattening Hierarchy (see Section 9.4.1)
- $t^{\Sigma_{state}}_{scc\mapsto scc}(\Sigma(\omega_{scc}))$: Removing States (see Section 9.4.2)

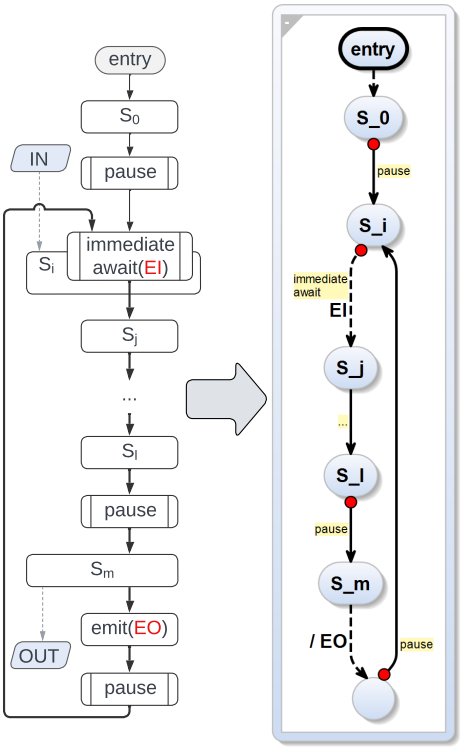
Translation Strategy:

This chapter introduces the translation strategies shown in Figure 9.2[1]. Figure 9.2a illustrates the translation strategy of a *Quartz* model without memory and Figure 9.2b shows the translation strategy of a *Quartz* model with memory, whose high-level runtime behavior was introduced in Chapter 4 for ST-based *Quartz* models and in Chapter 6 for FBD-based *Quartz* models. The translation strategy in this chapter leads to a graphical control-flow oriented SCChart that visualizes the control flow of the *Quartz* model (and the original FBD or ST model, respectively). In this context, a pattern-based code refactoring of the *Quartz* model is considered that (assuming the pattern is available) leads to a hierarchical control-flow oriented SCChart as an alternative view of the POU behavior. In addition, possible optimizations of the resulting SCChart are considered.

[1]The states can contain inner behavior which is considered, but hidden in the examples shown in this chapter.



(a) *Left: Quartz model, right: SCChart (both without memory)*



(b) *Left: Quartz model, right: SCChart (both with memory)*

Figure 9.2.: *Quartz-to-SCChart translation strategies: high-level view of the initial Quartz model and the resulting SCChart*

Challenges:

This approach leads to the following challenges:

1. Correct code refactoring of the initial *Quartz* model
2. Correct and complete translation of the considered *Quartz* constructs
3. Correct optimization of the resulting SCChart

9.2. Pattern-based *Quartz* Code Refactoring

This section introduces three different pattern-based *Quartz* code refactoring approaches for FBD-based *Quartz* models that are also applicable to ST-based *Quartz* models. The basic strategy is to split the conditions within a loop into nested loops that contain only those statements of the original conditions that are relevant to the current iteration [WS22]. Placing the statements at the end of a loop ensures that the statements are executed exactly once per iteration (even after an abort).

Definition 9.1 (Patterns – Quartz-to-Quartz). *Let* $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$ *be a possible* Quartz *sequence representing one of the following three patterns* $\vartheta \in \{1,$

$2,3\}$ *with instantaneous constructs and* $\Sigma_3(\omega_{qrz})$ *including the last statement that is equal to* $\sigma_{pause}(\omega_{qrz})$*, and each pattern considering two representations: (1) as a conditional expression and (2) as an* `if-else` *statement:*

- ***Pattern 1: Condition statement***

```
1 | loop{ Σ0;
2 |   x = λ1^b ? τ1 : τ2;
3 |   ... // add. assignments
4 |   Σ3; }
```

```
1 | // Σ1={x=τ1; ... }, Σ2= {x=τ2; ... }
2 | loop{ Σ0; if(λ1^b){ Σ1; }else{ Σ2; } Σ3; }
```

- ***Pattern 2: Nested condition statement within* else *branch***

```
1 | loop{ Σ0;
2 |   x = λ2^b ? τ4 : (λ1^b ? τ1 : τ2);
3 |   ... // add. assignments
4 |   Σ3; }
```

```
1 | // Σ1={x=τ1; ... }, Σ2={x=τ2; ... }, Σ4={x=τ4; ... }
2 | loop{ Σ0; if(λ2^b){ Σ4; }else{ if(λ1^b){ Σ1; }else{ Σ2; } } Σ3; }
```

- ***Pattern 3: Nested condition statement within* if *branch***

```
1 | loop{ Σ0;
2 |   x = λ2^b ? (λ1^b ? τ1 : τ2) : τ4;
3 |   ... // add. assignments
4 |   Σ3; }
```

```
1 | // Σ1={x=τ1; ... }, Σ2={x=τ2; ... }, Σ4={x=τ4; ... }
2 | loop{ Σ0; if(λ2^b){ if(λ1^b){ Σ1; }else{ Σ2; } }else{ Σ4; } Σ3; }
```

$\Sigma^{\vartheta}_{seq}(\omega_{qrz})$ *is refactored to* $\Sigma^{\vartheta}_{seq}(\omega_{qrz'})$ *using the translation function* $t^{\Sigma_{refact}}_{qrz\mapsto qrz}(\Sigma^{\vartheta}_{seq}(\omega_{qrz}))$*, which is described by Algorithm 25.*

Correctness

To check the correctness of Definition 9.1, the following lemma is used.

Lemma 9.1. *Let* $\vartheta \in \{1,2,3\}$ *and* $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$ *be a possible* Quartz *sequence of an infinite loop* $\sigma^{inf}_{loop}(\omega_{qrz})$ *containing instantaneous statements with the final statement equal to* $\sigma_{pause}(\omega_{qrz})$*. Then,* $t^{\Sigma_{refact}}_{qrz\mapsto qrz}(\Sigma^{\vartheta}_{seq}(\omega_{qrz}))$ *translates* $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$ *to* $\Sigma^{\vartheta}_{seq}(\omega_{qrz'})$ *as specified in Definition 9.1.* $\Sigma^{\vartheta}_{seq}(\omega_{qrz'})$ *conforms to the syntax rules of* $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$ *and preserves the SOS transition rules of* $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$*.*

Algorithm 25 Refactor *Quartz* code – Quartz-to-Quartz

Input: $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$
Output: $\Sigma^{\vartheta}_{seq}(\omega_{qrz'})$
Translation Function $t^{\Sigma_{refact}}_{qrz \mapsto qrz}(\Sigma^{\vartheta}_{seq}(\omega_{qrz}))$**:**

switch ϑ **do**

case *1 (Pattern 1)* **do**

$$\Sigma^{\vartheta}_{seq}(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} \texttt{loop\{} \\ \quad \texttt{immediate abort\{} \\ \quad\quad \texttt{loop\{}\ \Sigma_0(\omega_{qrz}); \Sigma_2(\omega_{qrz}); \Sigma_3(\omega_{qrz});\ \texttt{\}} \\ \quad \texttt{\}when(}\lambda^b_1(\omega_{qrz})\texttt{);} \\ \quad \Sigma_0(\omega_{qrz}); \Sigma_1(\omega_{qrz}); \Sigma_3(\omega_{qrz});\ \texttt{\}} \end{array} \right\}$$

end

case *2 (Pattern 2)* **do**

$$\Sigma^{\vartheta}_{seq}(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} \texttt{loop\{} \\ \quad \texttt{immediate abort\{} \\ \quad\quad \texttt{loop\{} \\ \quad\quad\quad \texttt{immediate abort\{} \\ \quad\quad\quad\quad \texttt{loop\{}\ \Sigma_0(\omega_{qrz}); \Sigma_2(\omega_{qrz}); \\ \quad\quad\quad\quad\quad \Sigma_3(\omega_{qrz});\ \texttt{\}} \\ \quad\quad\quad \texttt{\}when(}\lambda^b_1(\omega_{qrz})\texttt{);} \\ \quad\quad\quad \Sigma_0(\omega_{qrz}); \Sigma_1(\omega_{qrz}); \Sigma_3(\omega_{qrz}); \\ \quad\quad \texttt{\}when(}\lambda^b_2(\omega_{qrz})\texttt{);} \\ \quad \Sigma_0(\omega_{qrz}); \Sigma_4(\omega_{qrz}); \Sigma_3(\omega_{qrz});\ \texttt{\}} \end{array} \right\}$$

end

case *3 (Pattern 3)* **do**

$$\Sigma^{\vartheta}_{seq}(\omega_{qrz'}) \leftarrow \left\{ \begin{array}{l} \texttt{loop\{} \\ \quad \texttt{immediate abort\{} \\ \quad\quad \texttt{loop\{} \\ \quad\quad\quad \texttt{immediate abort\{} \\ \quad\quad\quad\quad \texttt{loop\{}\ \Sigma_0(\omega_{qrz}); \Sigma_2(\omega_{qrz}); \\ \quad\quad\quad\quad\quad \Sigma_3(\omega_{qrz});\ \texttt{\}} \\ \quad\quad\quad \texttt{\}when(}\lambda^b_1(\omega_{qrz})\texttt{);} \\ \quad\quad\quad \Sigma_0(\omega_{qrz}); \Sigma_1(\omega_{qrz}); \Sigma_3(\omega_{qrz}); \\ \quad\quad \texttt{\}when(!}\lambda^b_2(\omega_{qrz})\texttt{);} \\ \quad \Sigma_0(\omega_{qrz}); \Sigma_4(\omega_{qrz}); \Sigma_3(\omega_{qrz});\ \texttt{\}} \end{array} \right\}$$

end

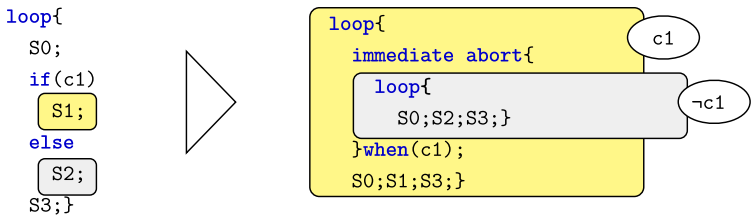
end

Proof The validity of Lemma 9.1 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma^{\vartheta}_{seq}(\omega_{qrz'})$ with the syntax rules of $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$ as specified in Section 3.3. Second, the preservation of semantic is checked for all scenarios of $\lambda^b_1(\omega_{qrz})$ and $\lambda^b_2(\omega_{qrz})$ by comparing the SOS transition rules of $\Sigma^{\vartheta}_{seq}(\omega_{qrz})$, $\sigma^{inf}_{loop}(\omega_{qrz})$, $\sigma^{imm}_{abort}(\omega_{qrz})$, and $\sigma_{pause}(\omega_{qrz})$ (see Section 3.5) with those of the initial constructs $\sigma^{\vartheta}_{seq}(\omega_{qrz})$,

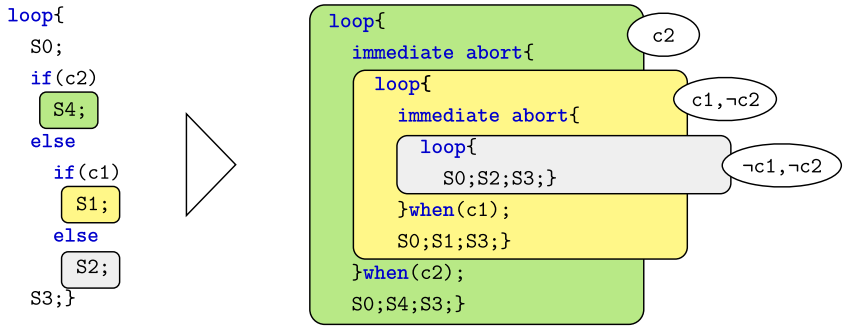
$\sigma^{inf}_{loop}(\omega_{qrz})$, $\sigma^{\vartheta}_{cond}(\omega_{qrz})$, and $\sigma_{pause}(\omega_{qrz})$ (see Section 3.3). Since strong aborts lead to a condition check before the *micro steps* are executed [Sch09] and the statements are instantaneous (except the last one of $\Sigma_3(\omega_{qrz})$), each *micro step* of the sequences is executed once per *macro step*, preserving the initial semantics.

Illustrative Example for Lemma 9.1

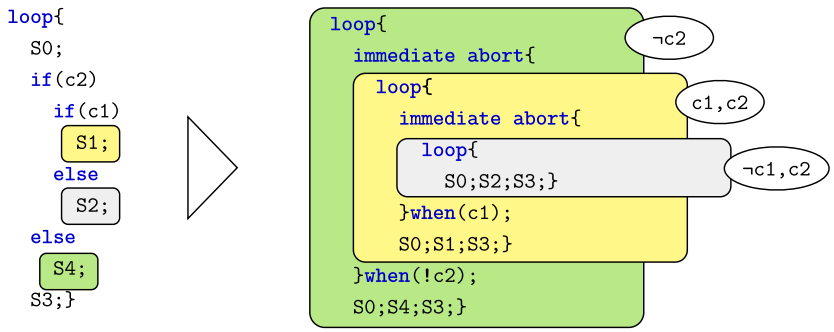
As an example, Figure 9.3 illustrates each of the three introduced patterns. In particular, Figure 9.3a illustrates Pattern 1, Figure 9.3b illustrates Pattern 2, and Figure 9.3c illustrates Pattern 3.



(a) *Pattern 1*



(b) *Pattern 2*



(c) *Pattern 3*

Figure 9.3.: *Illustration of the Quartz code pattern-based refactoring approach [WS22]*

9.3. From *Quartz* Models to SCCharts

This section defines the individual translation functions for synthesizing a graphical SCChart $\omega_{scc'} \in \Omega_{scc}$ from a *Quartz* model $\Omega_{qrz} \in \Omega_{qrz}$ and analyzes the theoretical correctness.

9.3.1. Model Declaration

This step covers the translation function for translating a *Quartz* model declaration $\delta_\omega(\omega_{qrz})$ to an SCChart declaration $\delta_\omega(\omega_{scc'})$.

Definition 9.2 (Model Declaration – Quartz-to-SCChart). *Let ω_{qrz} be the set of possible* Quartz *elements. $\delta_\omega(\omega_{qrz})$ is translated to $\delta_\omega(\omega_{scc'})$ using the translation function $t^{\delta_\omega}_{qrz\mapsto scc}(\delta_\omega(\omega_{qrz}))$, which is described by Algorithm 26.*

Algorithm 26 Translate model declaration – Quartz-to-SCChart

Input: $\delta_\omega(\omega_{qrz})$
Output: $\delta_\omega(\omega_{scc'})$
Translation Function $t^{\delta_\omega}_{qrz\mapsto scc}(\delta_\omega(\omega_{qrz}))$**:**

$$\delta_\omega(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{scchart } a_n(\omega_{qrz})\{ \\ \quad t^{\Delta_{idcl}}_{qrz\mapsto scc}(\Delta_{idcl}(\omega_{qrz})) \\ \quad t^{\Delta_{vdcl}}_{qrz\mapsto scc}(\Delta_{vdcl}(\omega_{qrz})) \\ \quad \texttt{region:} \\ \quad\quad t^{\Sigma}_{qrz\mapsto scc}(\omega_{qrz}) \\ \} \end{array} \right\}$$

Correctness

To check the correctness of Definition 9.2, the following lemma is used.

Lemma 9.2. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, $t^{\delta_\omega}_{qrz\mapsto scc}(\delta_\omega(\omega_{qrz}))$ translates $\delta_\omega(\omega_{qrz})$ to $\delta_\omega(\omega_{scc'})$ as specified in Definition 9.2. $\delta_\omega(\omega_{scc'})$ conforms to the syntax rules of $\delta_\omega(\omega_{scc})$ and preserves the semantics of $\delta_\omega(\omega_{qrz})$ regarding its termination behavior.*

Proof The validity of Lemma 9.2 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\delta_\omega(\omega_{scc'})$ with the syntax rules of $\delta_\omega(\omega_{scc})$ as specified in Section 3.5. Second, the preservation of semantics is checked by comparing $[\![\delta_\omega(\omega_{scc})]\!]_\xi$ (see Section 3.5) with $[\![\delta_\omega(\omega_{qrz})]\!]_\xi$ (see Section 3.3), where model imports are ignored in this chapter.

Illustrative Example for Lemma 9.2

As an example, Figure 9.4 shows the resulting textual and synthesized SCChart of the `ST_ALARM` model[2].

[2]Both models, *Quartz* and SCChart, are included in Appendix C.27 and H.1.

```
scchart ST_ALARM{
 input bool xSENSOR_L
 ...

 region:
   initial state S0
   immediate do ST_ALARM = ...
   final state S1
}
```

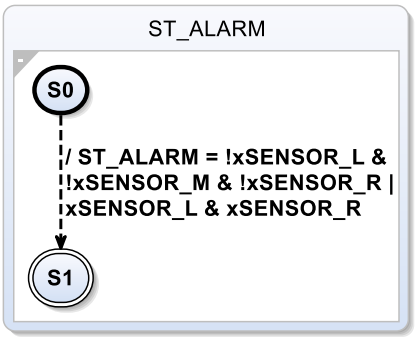


Figure 9.4.: *SCChart of the `ST_ALARM` example*

9.3.2. Interfaces

This step covers the translation function for translating *Quartz* model interfaces $\Delta_{idcl}(\omega_{qrz})$ to SCChart interfaces $\Delta_{idcl}(\omega_{scc'})$, where $\Delta_{idcl}(\omega_{qrz}) = \Delta_{in}(\omega_{qrz}) \cup \Delta_{out}(\omega_{qrz}) \cup \Delta_{inout}(\omega_{qrz})$.

Definition 9.3 (Interfaces – Quartz-to-SCChart). *$\Delta_{idcl}(\omega_{scc'})$ is derived from $\Delta_{idcl}(\omega_{qrz})$ using the translation function $t^{\Delta_{idcl}}_{qrz\mapsto scc}(\Delta_{idcl}(\omega_{qrz}))$, which is described by Algorithm 27.*

Algorithm 27 Translate interfaces – Quartz-to-SCChart

Input: $\Delta_{idcl}(\omega_{qrz})$
Output: $\Delta_{in}(\omega_{scc'})$, $\Delta_{out}(\omega_{scc'})$, $\Delta_{inout}(\omega_{scc'})$
Translation Function $t^{\Delta_{idcl}}_{qrz\mapsto scc}(\Delta_{idcl}(\omega_{qrz}))$**:**

```
forall e_i ∈ Δ_idcl(ω_qrz) do
    if e_i ∈ Δ_in(ω_qrz) then
        Δ_in(ω_scc') ← add input [signal]  t^α_{qrz↦scc}(α(e_i))  a_n(e_i)
    end
    if e_i ∈ Δ_out(ω_qrz) then
        Δ_out(ω_scc') ← add output [signal]  t^α_{qrz↦scc}(α(e_i))  a_n(e_i)
    end
    if e_i ∈ Δ_inout(ω_qrz) then
        Δ_inout(ω_scc') ← add input output  t^α_{qrz↦scc}(α(e_i))  a_n(e_i)
    end
end
                                  ▷ signal keyword is only required for event variables
```

Correctness

To check the correctness of Definition 9.3, the following lemma is used.

Lemma 9.3. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, for each interface $e_i \in \Delta_{idcl}(\omega_{qrz})$, $t^{\Delta_{idcl}}_{qrz\mapsto scc}(\Delta_{idcl}(\omega_{qrz}))$ extends $\Delta_{in}(\omega_{scc'})$, $\Delta_{out}(\omega_{scc'})$, or*

$\Delta_{inout}(\omega_{scc'})$ as specified in Definition 9.3. $\Delta_{in}(\omega_{scc'})$, $\Delta_{out}(\omega_{scc'})$, and $\Delta_{inout}(\omega_{scc'})$ conform to the syntax rules of $\Delta_{in}(\omega_{scc})$, $\Delta_{out}(\omega_{scc})$, and $\Delta_{inout}(\omega_{scc})$ regarding the storage class, data type, and name. $\Delta_{in}(\omega_{scc'})$, $\Delta_{out}(\omega_{scc'})$, and $\Delta_{inout}(\omega_{scc'})$ preserves the semantics of $\Delta_{in}(\omega_{qrz})$, $\Delta_{out}(\omega_{qrz})$, and $\Delta_{inout}(\omega_{qrz})$ regarding information flow and modifiability.

Proof The validity of Lemma 9.3 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{in}(\omega_{scc'})$, $\Delta_{out}(\omega_{scc'})$, and $\Delta_{inout}(\omega_{scc'})$ with the syntax rules of $\Delta_{in}(\omega_{scc})$, $\Delta_{out}(\omega_{scc})$, and $\Delta_{inout}(\omega_{scc})$ as specified in Section 3.5 using induction on the number of added interfaces $\Delta_{idcl}(\omega_{qrz})$:

1. **Base Case:** When $\Delta_{idcl}(\omega_{qrz}) = \varnothing$, there are no input variables, output variables, and inout variables to add, which trivially conforms to the syntax rules of $\Delta_{in}(\omega_{scc})$, $\Delta_{out}(\omega_{scc})$, and $\Delta_{inout}(\omega_{scc})$, since these sets remain unchanged.
2. **Induction Hypothesis:** Lemma 9.3 holds for any set of input variables, output variables, and inout variables.
3. **Inductive Step:** Adding an element to input variables, output variables, and inout variables results in an additional element in $\Delta_{in}(\omega_{scc'})$, $\Delta_{out}(\omega_{scc'})$, and $\Delta_{inout}(\omega_{scc'})$. Their syntax still conforms to the syntax rules of $\Delta_{in}(\omega_{scc})$, $\Delta_{out}(\omega_{scc})$, and $\Delta_{inout}(\omega_{scc})$.

Second, the semantic correctness regarding information flow and modifiability is checked by comparing $[\![\Delta_{in}(\omega_{scc})]\!]_\xi$, $[\![\Delta_{out}(\omega_{scc})]\!]_\xi$, and $[\![\Delta_{inout}(\omega_{scc})]\!]_\xi$ (see Section 3.5) with $[\![\Delta_{in}(\omega_{qrz})]\!]_\xi$, $[\![\Delta_{out}(\omega_{qrz})]\!]_\xi$, and $[\![\Delta_{inout}(\omega_{qrz})]\!]_\xi$ (see Section 3.3).

Illustrative Example for Lemma 9.3

As an example, below are the derived interfaces of the `ST_ALARM` *Quartz* model[3]:

```
1    input bool xSENSOR_L                    // added to Δin(ωscc')
2    input bool xSENSOR_M                    // added to Δin(ωscc')
3    input bool xSENSOR_R                    // added to Δin(ωscc')
```

9.3.3. Variables

This step covers the translation function for translating local *Quartz* model variables $\Delta_{vdcl}(\omega_{qrz})$ to local SCChart variables $\Delta_{vdcl}(\omega_{scc'})$), where $\Delta_{vdcl}(\omega_{qrz}) = \Delta_{local}(\omega_{qrz})$.

[3]Both models, *Quartz* and SCChart, are included in Appendix C.27 and H.1.

Definition 9.4 (Variables – Quartz-to-SCChart). *$\Delta_{vdcl}(\omega_{scc'})$ is derived from $\Delta_{vdcl}(\omega_{qrz})$ using the translation function $t^{\Delta_{vdcl}}_{qrz\mapsto scc}(\Delta_{local}(\omega_{qrz}))$, which is described by Algorithm 28.*

Algorithm 28 Translate variables – Quartz-to-SCChart

Input: $\Delta_{local}(\omega_{qrz})$
Output: $\Delta_{local}(\omega_{scc'})$
Translation Function $t^{\Delta_{vdcl}}_{qrz\mapsto scc}(\Delta_{local}(\omega_{qrz}))$**:**
  forall $e_{lVs} \in \Delta_{local}(\omega_{qrz})$ **do**
    $\Delta_{local}(\omega_{scc'}) \leftarrow$ add $t^{\alpha}_{qrz\mapsto scc}(\alpha(e_{lVs}))\ \ a_n(e_{lVs})$;
  end

Correctness

To check the correctness of Definition 9.4, the following lemma is used.

Lemma 9.4. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, for each local variable $e_{lVs} \in \Delta_{local}(\omega_{qrz}))$, $t^{\Delta_{vdcl}}_{qrz\mapsto scc}(\Delta_{local}(\omega_{qrz}))$ adds a local variable to $\Delta_{local}(\omega_{scc'})$ as specified in Definition 9.4. $\Delta_{local}(\omega_{scc'})$ conforms to the syntax rules of $\Delta_{local}(\omega_{scc})$ regarding the storage class, data type, and name. $\Delta_{local}(\omega_{scc'})$ preserves the semantics of $\Delta_{local}(\omega_{qrz})$ regarding modifiability and initialization.*

Proof The validity of Lemma 9.4 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Delta_{local}(\omega_{scc'})$ with the syntax rules of $\Delta_{local}(\omega_{scc})$ as specified in Section 3.5 using induction on the number of added variables $\Delta_{local}(\omega_{qrz})$:

1. **Base Case:** When $\Delta_{local}(\omega_{qrz}) = \varnothing$, there are no local variables to add, which trivially conforms to the syntax rules of $\Delta_{local}(\omega_{qrz})$, since this set remains unchanged.
2. **Induction Hypothesis:** Lemma 9.4 holds for any set of local variables.
3. **Inductive Step:** Adding a local variable results in an additional element in $\Delta_{local}(\omega_{scc'})$. Its syntax still conforms to the syntax rules of $\Delta_{local}(\omega_{scc})$.

Second, the semantic correctness regarding modifiability and initialization is checked by comparing $[\![\Delta_{local}(\omega_{scc})]\!]_{\xi}$ (see Section 3.5) with $[\![\Delta_{local}(\omega_{qrz})]\!]_{\xi}$ (see Section 3.3).

Illustrative Example for Lemma 9.4

As an example, below is a derived local variable of the `FBD_DATATYPES` *Quartz* model[4]:

```
1    bool A1                    // added to Δ_local(ω_scc')
```

9.3.4. Data Types and Fields

This step covers the translation function for translating *Quartz* data types and fields $\mathcal{A}^{[+]}(\omega_{qrz})$ to SCChart data types and fields $\mathcal{A}^{[+]}(\omega_{scc'})$).

Definition 9.5 (Data types and fields – Quartz-to-SCChart). *Let* $\alpha(\omega_{qrz})$ *be a considered* Quartz *data type and* $\alpha^{+}(\omega_{qrz})$ *be a* Quartz *model data field.* Quartz *data types and fields* $\mathcal{A}^{[+]}(\omega_{qrz})$ *are translated to SCChart data types and fields* $\mathcal{A}^{[+]}(\omega_{scc'})$ *using the translation function* $t^{\alpha}_{qrz\mapsto scc}(\alpha^{[+]}(\omega_{qrz}))$*, which is described by Algorithm 41 in Appendix M.0.1 that defines a straightforward mapping of data types.*

Correctness

To check the correctness of Definition 9.5, the following lemma is used.

Lemma 9.5. *Let* ω_{qrz} *be translated to* $\omega_{scc'}$ *and* bit vector*,* integer*,* floating point*, and* duration *be the considered data type categories* $\mathcal{A}^{[+]}(\omega_{qrz})$ *as specified in Section 3.3. Then,* $t^{\alpha}_{qrz\mapsto scc}(\alpha^{[+]}(\omega_{qrz}))$ *translates* $\alpha^{[+]}(\omega_{qrz})$ *to* $\alpha^{[+]}(\omega_{scc'})$ *as specified in Definition 9.5.* $\alpha^{[+]}(\omega_{scc'})$ *conforms to the syntax rules of* $\alpha^{[+]}(\omega_{qrz})$ *and preserves the semantics of* $\alpha^{[+]}(\omega_{qrz})$ *regarding boundaries, precision, resolution, and defaults (if applicable), with the following restrictions:*

- $\forall\alpha(\omega_{qrz}) : \alpha(\omega_{qrz}) \in \{\alpha^{byte}_{bv}(\omega_{qrz}), \alpha^{word}_{bv}(\omega_{qrz}), \texttt{nat}\{4294967296\}\}$*: Data types are not supported by internal SCChart data types*
- $\forall\alpha(\omega_{qrz}) : \alpha(\omega_{qrz}) \in \{\alpha^{time}_{dur}(\omega_{qrz}), \texttt{int}\{32768\}, \texttt{int}\{2147483648\}, \texttt{nat}\{65536\}\}$*: Boundaries are changed to those of* $\alpha^{int}_{i}(\omega_{scc})$ *(see Section 3.5)*

Proof The validity of Lemma 9.5 is proved as follows: First, the syntactic correctness is checked by comparing all resulting data types and fields $\mathcal{A}^{[+]}(\omega_{scc'})$ with the syntax rules of $\mathcal{A}^{[+]}(\omega_{scc})$ as specified in Section 3.5. Second, the semantic correctness regarding boundaries, precision, resolution, and defaults (if applicable) is checked by comparing $[\![\alpha^{[+]}(\omega_{scc})]\!]_{\xi}$ (see Section 3.5) with $[\![\alpha^{[+]}(\omega_{qrz})]\!]_{\xi}$ (see Section 3.3).

[4]Both models, *Quartz* and SCChart, are included in Appendix F.9 and H.8.

Illustrative Example for Lemma 9.5

Usage examples are given by illustrative examples of previous lemmas, such as Lemma 9.4.

9.3.5. Expressions

This step covers the translation function for translating expressions in *Quartz* models $\mathcal{T}(\omega_{qrz})$ to expressions in SCCharts $\mathcal{T}(\omega_{scc'})$.

Definition 9.6 (Expressions – Quartz-to-SCChart). *An expression in* Quartz *models* $\tau(\omega_{qrz}) \in \mathcal{T}(\omega_{qrz})$ *is translated to an expression in SCCharts* $\tau(\omega_{scc'}) \in \mathcal{T}(\omega_{scc'})$ *using the translation function* $t^{\tau}_{qrz \mapsto scc}(\tau(\omega_{qrz}))$*, which is described by Algorithm 42 in Appendix M.0.2.*

Correctness

To check the correctness of Definition 9.6, the following lemma is used.

Lemma 9.6. *Let* ω_{qrz} *be translated to* $\omega_{scc'}$ *and* miscellaneous*,* compare operators*,* arithmetic operators*,* conditional operator*, and* Boolean operators *be the considered expression categories* $\mathcal{T}(\omega_{qrz})$ *as specified in Section 3.3. Then, for each* $\tau(\omega_{qrz}) \in \mathcal{T}(\omega_{qrz})$*,* $t^{\tau}_{qrz \mapsto scc}(\tau(\omega_{qrz}))$ *translates* $\tau(\omega_{qrz})$ *to* $\tau(\omega_{scc'})$ *as specified in Definition 9.6.* $\tau(\omega_{scc'})$ *conforms to the syntax rules of* $\tau(\omega_{scc})$ *and preserves the semantics of* $\tau(\omega_{qrz})$ *regarding the type system and SOS transition rules, with the following restrictions:*

- $\forall \tau(\omega_{qrz}) : \tau \notin \{\tau^{inv}_{misc}, \tau^{expt}_{arith}\}$*, because they are not covered by the internal SCChart operators*

Proof The validity of Lemma 9.6 is proved as follows: First, syntactic correctness is checked by comparing the syntax of each resulting expression $\tau(\omega_{scc'})$ with the syntax rules of the corresponding expression $\tau(\omega_{scc})$ as specified in Section 3.5. Second, the semantic correctness regarding the type system and SOS transition rules is checked by comparing the semantics, type system, and SOS transition rule of each resulting expression $\tau(\omega_{scc})$ (see Section 3.5) with the semantics, type system, and SOS transition rule of the corresponding expression $\tau(\omega_{qrz})$ (see Section 3.3).

Illustrative Example for Lemma 9.6

As an example, below is the derived expression of the `ST_ALARM` *Quartz* model[5]:

```
1    (!(xSENSOR_L) & !(xSENSOR_M) & !(xSENSOR_R)) | (xSENSOR_L &
         xSENSOR_R)
```

[5]Both models, *Quartz* and SCChart, are included in Appendix C.27 and H.1.

9.3.6. Immediate Transitions

This step covers the translation function for translating an immediate transition pattern in *Quartz* $p_{nothing}(\omega_{qrz})$ to an immediate transition in SCCharts $\sigma_{nothing}(\omega_{scc'})$.

Definition 9.7 (Translation of immediate transitions – Quartz-to-SCChart). *An immediate transition in* Quartz $p_{nothing}(\omega_{qrz}) \in P_{nothing}(\omega_{qrz})$ *has the following pattern:*

```
1 | Σ1(p_nothing(ω_qrz)); σ_nothing(ω_qrz); Σ2(p_nothing(ω_qrz));
```

This pattern is translated to an immediate transition $\sigma_{nothing}(\omega_{scc'}) \in \Sigma_{nothing}(\omega_{scc'})$ *in SCCharts using the translation function* $t^{\Sigma_{nothing}}_{qrz \mapsto scc}(p_{nothing}(\omega_{qrz}))$, *which is described by Algorithm 29.*

Algorithm 29 Translate an immediate transition – Quartz-to-SCChart

Input: $p_{nothing}(\omega_{qrz})$
Output: $\sigma_{nothing}(\omega_{scc'})$
Translation Function $t^{\Sigma_{nothing}}_{qrz \mapsto scc}(p_{nothing}(\omega_{qrz}))$**:**

$$\sigma_{nothing}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(p_{nothing}(\omega_{qrz})) \\ \texttt{immediate go to } \Sigma_2(p_{nothing}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_2(p_{nothing}(\omega_{qrz})) \end{array} \right\}$$

▷ initial *and* final *identifiers are optional and are used for the first and last state*

Correctness

To check the correctness of Definition 9.7, the following lemma is used.

Lemma 9.7. *Let* ω_{qrz} *be translated to* $\omega_{scc'}$. *Then,* $t^{\Sigma_{nothing}}_{qrz \mapsto scc}(p_{nothing}(\omega_{qrz}))$ *translates* $p_{nothing}(\omega_{qrz})$ *to* $\sigma_{nothing}(\omega_{scc'})$ *as specified in Definition 9.7.* $\sigma_{nothing}(\omega_{scc'})$ *conforms to the syntax rules of* $\sigma_{nothing}(\omega_{scc})$ *and preserves the SOS transition rules of* $p_{nothing}(\omega_{qrz})$.

Proof The validity of Lemma 9.7 is proved as follows: First, syntactic correctness is checked by comparing the syntax of $\sigma_{nothing}(\omega_{scc'})$ with the syntax rules of $\sigma_{nothing}(\omega_{scc})$ as specified in Section 3.5. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma_{nothing}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\Sigma_1(\sigma_{nothing}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$, $\sigma_{nothing}(\omega_{qrz})$, and $\Sigma_2(\sigma_{nothing}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$ (see Section 3.3).

```
...
  initial state S1
  immediate go to S2
  final state S2
...
```

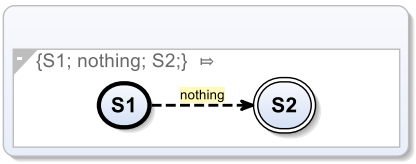


Figure 9.5.: *SCChart of* Quartz *sequence* {`S1; nothing; S2;`} *[WS22]*

Illustrative Example for Lemma 9.7

As an example, Figure 9.5 shows an immediate transition from state `S1` to state `S2`. `S1` is entered and left in the same *macro step*, because the nothing statement does not change any variables or stop control flow [WS22].

9.3.7. Await

This step covers the translation function for translating await statement patterns in *Quartz* $P^{\vartheta}_{await}(\omega_{qrz})$ to await transitions in SCCharts $\Sigma^{\vartheta}_{await}(\omega_{scc'})$.

Definition 9.8 (Translation of await statements – Quartz-to-SCChart). *A sequence containing an await statement in* Quartz $p^{\vartheta}_{await}(\omega_{qrz}) \in P^{\vartheta}_{await}(\omega_{qrz})$ *has one of the following patterns:*

- ***Await*** *($\vartheta = reg$)*

$$\Sigma_1(p^{reg}_{await}(\omega_{qrz}));\ \sigma^{reg}_{await}(\omega_{qrz});\ \Sigma_2(p^{reg}_{await}(\omega_{qrz}));$$

- ***Immediate await*** *($\vartheta = imm$)*

$$\Sigma_1(p^{imm}_{await}(\omega_{qrz}));\ \sigma^{imm}_{await}(\omega_{qrz});\ \Sigma_2(p^{imm}_{await}(\omega_{qrz}));$$

Each of these patterns is translated into a corresponding await transition in SCCharts $\sigma^{\vartheta}_{await}(\omega_{scc'}) \in \Sigma^{\vartheta}_{await}(\omega_{scc'})$ *using the translation function* $t^{\Sigma_{await}}_{qrz \mapsto scc}(p^{\vartheta}_{await}(\omega_{qrz}))$*, which is described by Algorithm 30.*

Correctness

To check the correctness of Definition 9.8, the following lemma is used.

Lemma 9.8. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, $t^{\Sigma_{await}}_{qrz \mapsto scc}(p^{\vartheta}_{await}(\omega_{qrz}))$ translates $p^{\vartheta}_{await}(\omega_{qrz})$ to $\sigma^{\vartheta}_{await}(\omega_{scc'})$ as specified in Definition 9.8. $\sigma^{\vartheta}_{await}(\omega_{scc'})$ conforms to the syntax rules of $\sigma^{\vartheta}_{await}(\omega_{scc})$ and preserves the SOS transition rules of $p^{\vartheta}_{await}(\omega_{qrz})$.*

Proof The validity of Lemma 9.8 is proved as follows: First, syntactic correctness is checked by comparing the syntax of $\sigma^{\vartheta}_{await}(\omega_{scc'})$ with the syntax rules of $\sigma^{\vartheta}_{await}(\omega_{scc})$ as specified in Section 3.5. Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma^{\vartheta}_{await}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\Sigma_1(p^{\vartheta}_{await}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$, $\sigma^{\vartheta}_{await}(\omega_{qrz})$, and $\Sigma_2(p^{\vartheta}_{await}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$ (see Section 3.3).

Algorithm 30 Translate await statement – Quartz-to-SCChart

Input: $p^{\vartheta}_{await}(\omega_{qrz})$
Output: $\sigma^{\vartheta}_{await}(\omega_{scc'})$
Translation Function $t^{\Sigma_{await}}_{qrz \mapsto scc}(p^{\vartheta}_{await}(\omega_{qrz}))$:

if $\vartheta = reg$ **then**

$$\sigma^{reg}_{await}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(p^{reg}_{await}(\omega_{qrz})) \\ \texttt{if } \lambda^b(\sigma^{reg}_{await}(\omega_{qrz})) \texttt{ abort to} \\ \quad \Sigma_2(p^{reg}_{await}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_2(p^{reg}_{await}(\omega_{qrz})) \end{array} \right\}$$

end
if $\vartheta = imm$ **then**

$$\sigma^{imm}_{await}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(p^{imm}_{await}(\omega_{qrz})) \\ \texttt{immediate if } \lambda^b(\sigma^{imm}_{await}(\omega_{qrz})) \texttt{ abort} \\ \quad \texttt{to } \Sigma_2(p^{imm}_{await}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_2(p^{imm}_{await}(\omega_{qrz})) \end{array} \right\}$$

end
▷ initial *and* final *identifiers are optional and are used for the first and last state*

Illustrative Examples for Lemma 9.8

As an example, Figure 9.6 shows a regular await statement, where the transition from state `S1` to state `S2` becomes active in the next *macro step* after `S1` is entered. In contrast, Figure 9.7 shows an immediate await statement, where the transition from state `S1` to state `S2` becomes active in the same *macro step* when `S1` is entered [WS22] (if Boolean condition `a` is true).

```
1  ...
2    initial state S1
3    if a abort to S2
4    final state S2
5  ...
```

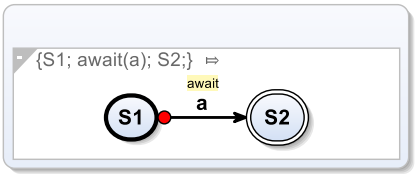


Figure 9.6.: *SCChart of* Quartz *sequence* {`S1; await(a); S2;`} *[WS22]*

```
1  ...
2    initial state S1
3    immediate if a abort to S2
4    final state S2
5  ...
```

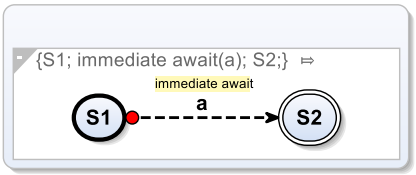


Figure 9.7.: *SCChart of* Quartz *sequence* {`S1; immediate await(a); S2;`} *[WS22]*

9.3.8. Pause

This step covers the translation function for translating `pause` statement patterns in *Quartz* $P_{pause}(\omega_{qrz})$ to *pause* transitions in SCCharts $\Sigma_{pause}(\omega_{scc'})$.

Definition 9.9 (Translation of pause statements – Quartz-to-SCChart). *A* `pause` *statement* $p_{pause}(\omega_{qrz}) \in P_{pause}(\omega_{qrz})$ *is translated to a* pause *transition in SCCharts* $\sigma_{pause}(\omega_{scc'}) \in \Sigma_{pause}(\omega_{scc'})$ *using* $t^{\Sigma_{pause}}_{qrz \mapsto scc}(p_{pause}(\omega_{qrz}))$, *where* $t^{\Sigma_{pause}}_{qrz \mapsto scc}(p_{pause}(\omega_{qrz})) = t^{\Sigma_{await}}_{qrz \mapsto scc}(p^{reg}_{await}(\omega_{qrz})) \iff \lambda^b(\sigma^{reg}_{await}(\omega_{qrz})) = true$.

Correctness

To check the correctness of Definition 9.9, the following lemma is used.

Lemma 9.9. *Lemma 9.8 is also valid for* pause *statements, if* $\lambda^b(\sigma^{reg}_{await}(\omega_{qrz})) = true$.

Proof Lemma 9.9 is valid, because a *pause* statement is never instantaneous. It consumes a logical unit of time. Therefore, both *Quartz* statements, `pause` and `wait(true)`, are equivalent, as confirmed by Schneider [Sch09] and demonstrated in the example below [WS22].

Illustrative Examples for Lemma 9.9

As an example, Figure 9.8 shows a `await(true)` transition in the resulting SCChart. In contrast, Figure 9.9 shows a *pause* transition in the resulting SCChart. Both resulting SCCharts are equivalent.

```
1    ...
2    initial state S1
3    if true abort to S2
4    final state S2
5    ...
```

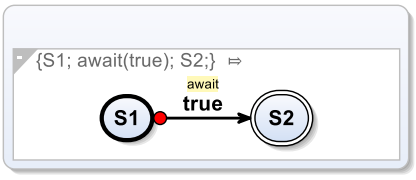


Figure 9.8.: *SCChart of* Quartz *sequence* {`S1; await(true); S2;`} *[WS22]*

```
1    ...
2    initial state S1
3    abort to S2
4    final state S2
5    ...
```

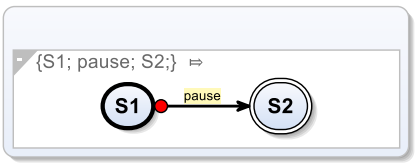


Figure 9.9.: *SCChart of* Quartz *sequence* {`S1; pause; S2;`} *[WS22]*

9.3.9. Assignments

This step covers the translation function for translating assignment patterns in *Quartz* models $P^{\vartheta}_{ass}(\omega_{qrz})$ to sequences including assignments in SCCharts $\Sigma_{seq}(\omega_{scc'})$.

Definition 9.10 (Translation of assignments – Quartz-to-SCChart). *A sequence including an immediate assignment and a delayed assignment in* Quartz $p^{\vartheta}_{ass}(\omega_{qrz}) \in P^{\vartheta}_{ass}(\omega_{qrz})$ *has one of the following patterns:*

- ***Immediate Assignment*** *($\vartheta = imm$)*

1 | $\sigma^{imm}_{ass}(\omega_{qrz})\,;\ \Sigma_1(p^{imm}_{ass}(\omega_{qrz}))\,;$

- ***Delayed Assignment*** *($\vartheta = del$)*

1 | $\sigma^{del}_{ass}(\omega_{qrz})\,;\ \sigma_{pause}(\omega_{qrz}))\,;\ \Sigma_1(p^{del}_{ass}(\omega_{qrz}))\,;$

Each of these patterns is translated to a corresponding sequence including assignment in SCCharts $\Sigma_{seq}(\omega_{scc'})$ using the translation function $t^{\Sigma_{ass}}_{qrz\mapsto scc}(p^{\vartheta}_{ass}(\omega_{qrz}))$, which is described by Algorithm 31.

Algorithm 31 Translate assignment – Quartz-to-SCChart

Input: $p^{\vartheta}_{ass}(\omega_{qrz})$
Output: $\Sigma_{seq}(\omega_{scc'})$
Translation Function $t^{\Sigma_{ass}}_{qrz\mapsto scc}(p^{\vartheta}_{ass}(\omega_{qrz}))$**:**

if $\vartheta = imm$ **then**

$$\Sigma_{seq}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{immediate do } \sigma^{imm}_{ass}(\omega_{qrz}) \texttt{ go to } \Sigma_1(p^{imm}_{ass}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_1(p^{imm}_{ass}(\omega_{qrz})) \end{array} \right\}$$

end

if $\vartheta = del$ **then**

$$\Sigma_{seq}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{state } \Sigma_{p,1} \texttt{ "pause" \{} \\ \quad \texttt{initial state } \Sigma_{00} \texttt{ ""} \\ \quad \texttt{do } lhs(\sigma^{del}_{ass}(\omega_{qrz}))) \texttt{ = pre(} rhs(\sigma^{del}_{ass}(\omega_{qrz})))) \\ \quad\quad \texttt{abort to } \Sigma_{01} \\ \quad \texttt{final state } \Sigma_{01} \texttt{ ""} \\ \texttt{\}} \\ \texttt{immediate [do } rhs(p^{del}_{ass}(\omega_{qrz})) \texttt{ = ...] join} \\ \quad \texttt{to } \Sigma_1(p^{del}_{ass}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_1(p^{del}_{ass}(\omega_{qrz})) \end{array} \right\}$$

▷ *$lhs(\sigma^{del}_{ass}(\omega_{qrz})))$ without* `next` *keyword*

end

▷ initial *and* final *identifiers are optional and are used for the first and last state*

Correctness

To check the correctness of Definition 9.10, the following lemmas are used.

Lemma 9.10. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, $t^{\Sigma_{ass}}_{qrz \mapsto scc}(p^{\vartheta}_{ass}(\omega_{qrz}))$ translates $p^{imm}_{ass}(\omega_{qrz})$ to $\Sigma_{seq}(\omega_{scc'})$ as specified in Definition 9.10. $\Sigma_{seq}(\omega_{scc'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{scc})$ and preserves the SOS transition rules of $p^{imm}_{ass}(\omega_{qrz})$.*

Proof The validity of Lemma 9.10 is proved as follows: First, the syntactic correctness is checked by comparing the syntax of $\Sigma_{seq}(\omega_{scc'})$ with the syntax rules of $\Sigma_{seq}(\omega_{scc})$ as specified in Section 3.5, considering the *do* prefix of assignments. Second, the semantic correctness is checked by comparing the SOS transition rules of $\Sigma_{seq}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\sigma^{imm}_{ass}(\omega_{qrz})$ and $\Sigma_1(p^{imm}_{ass}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$ (see Section 3.3).

Illustrative Examples for Lemma 9.10

As an example, Figure 9.10 shows an immediate assignment, where `a` is instantaneously set to `b` [WS22].

```
1  ...
2    initial state S1
3    immediate do a=b go to S2
4    final state S2
5  ...
```

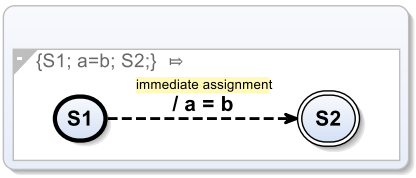


Figure 9.10.: *SCChart of* Quartz *sequence* {`S1; a=b; S2;`} *[WS22]*

Lemma 9.11. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, $t^{\Sigma_{ass}}_{qrz \mapsto scc}(p^{\vartheta}_{ass}(\omega_{qrz}))$ translates $p^{del}_{ass}(\omega_{qrz})$ to $\Sigma_{seq}(\omega_{scc'})$ as specified in Definition 9.10. $\Sigma_{seq}(\omega_{scc'})$ conforms to the syntax rules of $\Sigma_{seq}(\omega_{scc})$ and preserves the SOS transition rules of $p^{del}_{ass}(\omega_{qrz})$.*

Proof The validity of Lemma 9.11 is proved as follows: First, the syntactic correctness is checked by comparing the syntax of $\Sigma_{seq}(\omega_{scc'})$ with the syntax rules of $\Sigma_{seq}(\omega_{scc})$ as specified in Section 3.5, considering the `pre` operator in SCCharts applied in [WS22] and *do* prefix of assignments. Second, the semantic correctness is checked by comparing the SOS transition rules of $\Sigma_{seq}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\sigma^{del}_{ass}(\omega_{qrz})$, $\sigma_{pause}(\omega_{qrz})$, and $\Sigma_1(p^{del}_{ass}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$ (see Section 3.3).

Illustrative Examples for Lemma 9.11

As an example, Figure 9.11 shows a delayed assignment, where `a` is instantaneously set to `1` and in the next *macro step*, `b` is set to `1`, although `a` is set to `2` in the same *macro step* [WS22].

```
1  ...
2    initial state S1
3    immediate do a =1 go to S
4    state S "pause" {
5      initial state S0 ""
6      do b = pre(a) abort to
           S1
7      final state S1 ""
8    }
9    immediate do a=2 join to
         S2
10   final state S2
11 ...
```

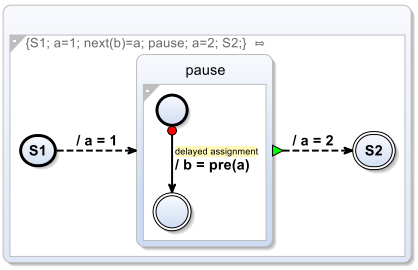


Figure 9.11.: *SCChart of* Quartz *sequence* {`S1; a=1; next(b)=a; pause; a=2; S2;`} *[WS22]*

9.3.10. Synchronous Concurrency

This step covers the translation function for translating synchronous concurrency in *Quartz* $\Sigma_{conc}(\omega_{qrz})$ to parallel regions in SCCharts $\Sigma_{conc}(\omega_{scc'})$.

Definition 9.11 (Translation of synchronous concurrency – Quartz-to-SCChart). *Synchronous concurrency in* Quartz $\Sigma_{conc}(\omega_{qrz})$ *has the following pattern:*

$\Sigma_1(\omega_{qrz})$; // ... // $\Sigma_n(\omega_{qrz})$;

This pattern is translated to parallel regions in SCCharts $\Sigma_{conc}(\omega_{scc'})$ *using the translation function* $t^{\Sigma_{conc}}_{qrz \mapsto scc}(\Sigma_{conc}(\omega_{qrz}))$*, which is described by Algorithm 32.*

Algorithm 32 Translate synchronous concurrency – Quartz-to-SCChart

Input: $\Sigma_{conc}(\omega_{qrz})$
Output: $\Sigma_{conc}(\omega_{scc'})$
Translation Function $t^{\Sigma_{conc}}_{qrz \mapsto scc}(\Sigma_{conc}(\omega_{qrz}))$**:**

$$\Sigma_{conc}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{region:} \\ \quad \texttt{initial state } \Sigma_1(\Sigma_{conc}(\omega_{qrz})) \\ \ldots \\ \texttt{region:} \\ \quad \texttt{initial state } \Sigma_n(\Sigma_{conc}(\omega_{qrz})) \end{array} \right\}$$

Correctness

To check the correctness of Definition 9.11, the following lemma is used.

Lemma 9.12. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, $t^{\Sigma_{conc}}_{qrz \mapsto scc}(\Sigma_{conc}(\omega_{qrz}))$ translates $\Sigma_{conc}(\omega_{qrz})$ to $\Sigma_{conc}(\omega_{scc'})$ as specified in Definition 9.11. $\Sigma_{conc}(\omega_{scc'})$ conforms to the syntax rules of $\Sigma_{conc}(\omega_{scc})$ and preserves the SOS transition rules of $\Sigma_{conc}(\omega_{qrz})$.*

Proof The validity of Lemma 9.12 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{conc}(\omega_{scc'})$ with the syntax rules of $\Sigma_{conc}(\omega_{scc})$ as specified in Section 3.5, where $n = 2$. Second, the semantic correctness is proved by comparing the SOS transition rules of $\Sigma_{conc}(\omega_{scc})$ (see Section 3.5) with those of $\Sigma_{conc}(\omega_{qrz})$ (see Section 3.3), where $n = 2$.

Illustrative Example for Lemma 9.12

As an example, Figure 9.12 shows synchronous concurrency of states `S1`, `S2`, `S3`, and `Sn`. At the *macro steps*, the statements in the regions are synchronized and can interact. Any abort statements will affect all parallel regions. The synchronous concurrency terminates as soon as the last of the parallel regions terminates [WS22].

```
1  ...
2    region:
3      initial state S1
4    region:
5      initial state S2
6    region:
7      initial state S3
8    region:
9      initial state Sn
10 ...
```

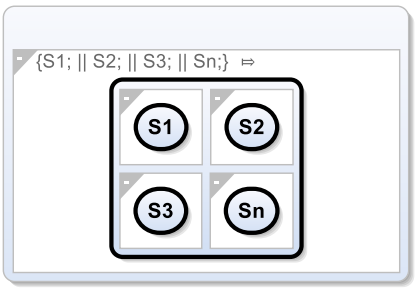


Figure 9.12.: *SCChart of* Quartz *sequence* {`S1; || S2; || S3; || Sn;`} *[WS22]*

9.3.11. Loops

This step covers the translation function for translating loop variants in *Quartz* $P^{\vartheta}_{loop}(\omega_{qrz})$ to loop variants in SCCharts $\Sigma^{\vartheta}_{loop}(\omega_{scc'})$.

Definition 9.12 (Translation of loops – Quartz-to-SCChart). *A loop in* Quartz *has one of the following patterns $p^{\vartheta}_{loop}(\omega_{qrz}) \in P^{\vartheta}_{loop}(\omega_{qrz})$:*

- ***Head-controlled loop*** *($\vartheta = head$, $\Sigma_1(p^{head}_{loop}(\omega_{qrz})) := \Sigma(\sigma^{head}_{loop}(\omega_{qrz}))$)*

1 | $\sigma^{head}_{loop}(\omega_{qrz})$; $\Sigma_2(p^{head}_{loop}(\omega_{qrz}))$;

- ***Foot-controlled loop*** *($\vartheta = foot$, $\Sigma_1(p^{foot}_{loop}(\omega_{qrz})) := \Sigma(\sigma^{foot}_{loop}(\omega_{qrz}))$)*

1 | $\sigma^{foot}_{loop}(\omega_{qrz})$; $\Sigma_2(p^{foot}_{loop}(\omega_{qrz}))$;

- ***Infinite loop*** *($\vartheta = inf$, $\Sigma_1(p^{inf}_{loop}(\omega_{qrz})) := \Sigma(\sigma^{inf}_{loop}(\omega_{qrz}))$)*

1 | $\sigma^{inf}_{loop}(\omega_{qrz})$; $\Sigma_2(p^{foot}_{loop}(\omega_{qrz}))$;

Each of these patterns is translated to a corresponding loop in SCCharts $\sigma^{\vartheta}_{loop}(\omega_{scc'}) \in \Sigma^{\vartheta}_{loop}(\omega_{scc'})$ using the translation function $t^{\Sigma_{loop}}_{qrz \mapsto scc}(p^{\vartheta}_{loop}(\omega_{qrz}))$, which is described by Algorithm 33.

Correctness

To check the correctness of Definition 9.12, the following lemma is used.

Lemma 9.13. *Let ω_{qrz} be translated to $\omega_{scc'}$. Then, $t^{\Sigma_{loop}}_{qrz \mapsto scc}(p^{\vartheta}_{loop}(\omega_{qrz}))$ translates $p^{\vartheta}_{loop}(\omega_{qrz})$ to $\sigma^{\vartheta}_{loop}(\omega_{scc'})$ as specified in Definition 9.12. $\sigma^{\vartheta}_{loop}(\omega_{scc'})$ conforms to the syntax rules of $\sigma^{\vartheta}_{loop}(\omega_{scc})$ and preserves the SOS transition rules of $p^{\vartheta}_{loop}(\omega_{qrz})$.*

Proof The validity of Lemma 9.13 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma^{\vartheta}_{loop}(\omega_{scc'})$ with the syntax rules of $\sigma^{\vartheta}_{loop}(\omega_{scc})$ as specified in Section 3.5, considering nesting of statements. Second, the semantic correctness is checked by comparing the SOS transition rules of $\Sigma^{\vartheta}_{loop}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\sigma^{\vartheta}_{loop}(\omega_{qrz})$ and $\Sigma_2(p^{\vartheta}_{loop}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$ (see Section 3.3).

Illustrative Examples for Lemma 9.13

As an example, Figure 9.13 shows a head-controlled loop $\sigma^{head}_{loop}(\omega_{qrz})$, where `S1` immediately switches to a final state where a Boolean condition `a` is checked before the other statements `S1` are executed. In contrast, Figure 9.14 shows a foot-controlled loop $\sigma^{foot}_{loop}(\omega_{qrz})$, where the statement `S1` is executed and the Boolean condition `a` is ignored. After `S1` has been terminated, `a` is checked. If `a` is `true` then `S1` is executed again, otherwise `S1` is not executed again [WS22]. Figure 9.14 illustrates an infinite loop, that executes statement `S1` without checking a Boolean condition. Consequently, this loop variant represents a special case of the foot-controlled loop $\sigma^{foot}_{loop}(\omega_{qrz})$ for which it can be assumed that the termination condition is never true[6] [WS22].

[6] `!a = false`, because `a` is always `true`

Algorithm 33 Translate loop – Quartz-to-SCChart

Input: $p^{\vartheta}_{loop}(\omega_{qrz})$

Output: $\sigma^{\vartheta}_{loop}(\omega_{scc'})$

Translation Function $t^{\Sigma_{loop}}_{qrz\mapsto scc}(p^{\vartheta}_{loop}(\omega_{qrz}))$**:**

if ϑ = *head* **then**

$$\sigma^{head}_{loop}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{head}_{loop}(\omega_{qrz}))\{ \\ \quad \texttt{initial state } \Sigma_1 \\ \quad \texttt{immediate go to } \Sigma_2 \\ \quad \texttt{final state } \Sigma_2 \\ \quad \texttt{immediate if } \lambda^b(\sigma^{head}_{loop}(\omega_{qrz}) \texttt{ go to } \ldots \\ \quad \ldots \\ \} \\ \texttt{immediate if !}(\lambda^b(\sigma^{head}_{loop}(\omega_{qrz}))) \texttt{ join} \\ \quad \texttt{to } \Sigma_2(p^{head}_{loop}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_2(p^{head}_{loop}(\omega_{qrz})) \end{array} \right.$$

end

if ϑ = *foot* **then**

$$\sigma^{foot}_{loop}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{foot}_{loop}(\omega_{qrz}))\{ \\ \quad \texttt{initial state } \Sigma_1 \\ \quad \ldots \\ \quad \texttt{final state } \Sigma_n \\ \quad \texttt{immediate if } \lambda^b(\sigma^{foot}_{loop}(\omega_{qrz})) \texttt{ go to } \Sigma_1 \\ \} \\ \texttt{immediate if !}(\lambda^b(\sigma^{foot}_{loop}(\omega_{qrz}))) \texttt{ join} \\ \quad \texttt{to } \Sigma_2(p^{foot}_{loop}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_2(p^{foot}_{loop}(\omega_{qrz})) \end{array} \right.$$

end

if ϑ = *inf* **then**

$\sigma^{inf}_{loop}(\omega_{scc'}) \leftarrow \sigma^{foot}_{loop}(\omega_{scc'}) \iff \lambda^b(\sigma^{foot}_{loop}(\omega_{qrz})) = true$

end

▷ initial *and* final *identifiers are optional and are used for the first and last state*

9.3.12. Halt

This step covers the translation function for translating *halt* statements in *Quartz* $\Sigma_{halt}(\omega_{qrz})$ to *halt* statements in SCCharts $\Sigma_{halt}(\omega_{scc'})$.

Definition 9.13 (Translation of halt statements – Quartz-to-SCChart). *A* halt *statement in* Quartz $\sigma_{halt}(\omega_{qrz}) \in \Sigma_{halt}(\omega_{qrz})$ *is translated to a* halt *statement in SCCharts* $\sigma_{halt}(\omega_{scc'}) \in \Sigma_{halt}(\omega_{scc'})$ *using* $t^{\Sigma_{halt}}_{qrz\mapsto scc}(\sigma_{halt}(\omega_{qrz}))$*, where* $t^{\Sigma_{halt}}_{qrz\mapsto scc}(\sigma_{halt}(\omega_{qrz})) = t^{\Sigma_{loop}}_{qrz\mapsto scc}(\sigma^{inf}_{loop}(\omega_{qrz})) \iff \Sigma(\sigma^{inf}_{loop}(\omega_{qrz})) = pause$*.*

```
...
initial state S1{
  initial final state S1 ""
  immediate if a go to S2
  ...
}
immediate if !a join to S2
final state S2
...
```

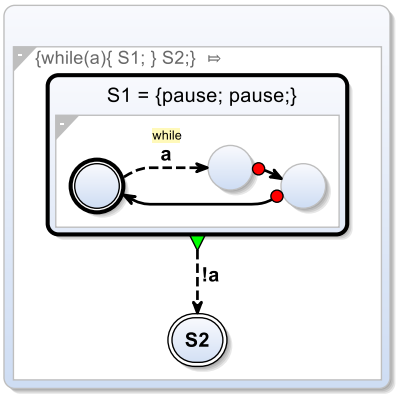


Figure 9.13.: *SCChart of* Quartz *sequence* {`while(a){S1;} S2;`} *[WS22]*

```
...
initial state S1{
  initial state S1 ""
  ...
  final state S3 ""
  immediate if a go to S1
}
immediate if !a join to S2
final state S2
...
```

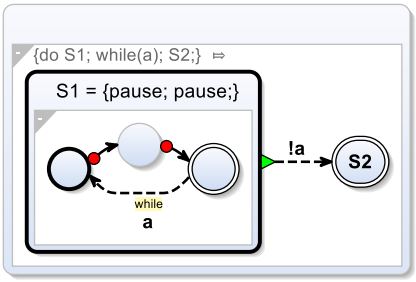


Figure 9.14.: *SCChart of* Quartz *sequence* {`do S1; while(a); S2;`} *[WS22]*

```
...
initial state S1{
  initial state S1
  ...
  final state S3 ""
  immediate go to S1
}
...
```

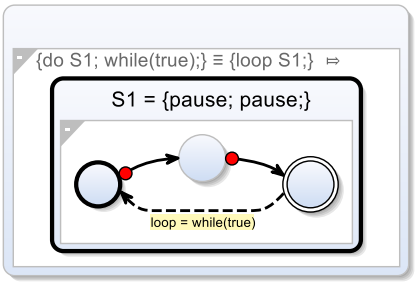


Figure 9.15.: *SCChart of* Quartz *sequence* {`loop S;`} *[WS22]*

Correctness

To check the correctness of Definition 9.13, the following lemma is used.

Lemma 9.14. *Lemma 9.13 is also valid for* halt *statements* $\Sigma_{halt}(\omega_{qrz})$, *if* $\Sigma(\sigma^{inf}_{loop}(\omega_{qrz})) = pause$.

Proof Lemma 9.14 is valid, because the *halt* statement represents an infinite loop executing a `pause` statement that consumes one logical unit of time per

iteration. Consequently, the *halt* statement never terminates [WS22]. Therefore, both *Quartz* statements, `halt` and `loop{pause;}`, are equivalent, as demonstrated in the illustrative example below and confirmed by Schneider [Sch09].

Illustrative Examples for Lemma 9.14

As an example, Figure 9.16 shows a *halt* statement realized as infinite loop. In contrast, Figure 9.17 shows a *halt* statement with a resulting state that never terminates [WS22].

```
1    ...
2    initial state S1 ""
3    abort to S2
4    final state S2 ""
5    immediate go to S1
6    ...
```

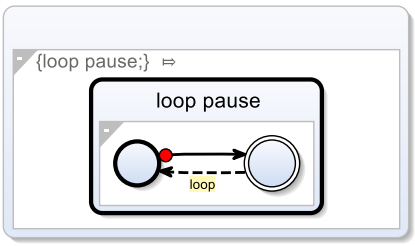


Figure 9.16.: *SCChart of* Quartz *sequence* {`loop{pause;}`} *[WS22]*

```
1    ...
2    initial state S1 ""
3    ...
```

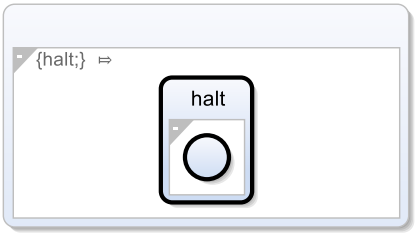


Figure 9.17.: *SCChart of* Quartz *sequence* {`halt;`} *[WS22]*

9.3.13. Abort

This step covers the translation function for translating abort statement patterns in *Quartz* $P^{\vartheta}_{abort}(\omega_{qrz})$ to abort transitions in SCCharts $\Sigma^{\vartheta}_{abort}(\omega_{scc'})$.

Definition 9.14 (Translation of abort statements – Quartz-to-SCChart). *An abort statement in* Quartz *has one of the following patterns* $p^{\vartheta}_{abort}(\omega_{qrz}) \in P^{\vartheta}_{abort}(\omega_{qrz})$*:*

- ***Abort*** *(*$\vartheta = reg$, $\Sigma_1(p^{reg}_{abort}(\omega_{qrz})) := \Sigma(\sigma^{reg}_{abort}(\omega_{qrz}))$*)*

  $\sigma^{reg}_{abort}(\omega_{qrz})$; $\Sigma_2(p^{reg}_{abort}(\omega_{qrz}))$;

- ***Immediate abort*** *(*$\vartheta = imm$, $\Sigma_1(p^{imm}_{abort}(\omega_{qrz})) := \Sigma(\sigma^{imm}_{abort}(\omega_{qrz}))$*)*

  $\sigma^{imm}_{abort}(\omega_{qrz})$; $\Sigma_2(p^{imm}_{abort}(\omega_{qrz}))$;

Each of these patterns is translated to a corresponding abort transition in SCCharts $\sigma^{\vartheta}_{abort}(\omega_{scc'}) \in \Sigma^{\vartheta}_{abort}(\omega_{scc'})$ *using the translation function* $t^{\Sigma_{abort}}_{qrz \mapsto scc}(p^{\vartheta}_{abort}(\omega_{qrz}))$*, which is described by Algorithm 34.*

Algorithm 34 Translate abort – Quartz-to-SCChart

Input: $p^{\vartheta}_{abort}(\omega_{qrz})$
Output: $\sigma^{\vartheta}_{abort}(\omega_{scc'})$
Translation Function $t^{\Sigma_{abort}}_{qrz \mapsto scc}(p^{\vartheta}_{abort}(\omega_{qrz}))$**:**

if $\vartheta = reg$ **then**

$$\sigma^{reg}_{abort}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(p^{reg}_{abort}(\omega_{qrz})) \\ \texttt{if } \lambda^b(\sigma^{reg}_{abort}(\omega_{qrz})) \texttt{ abort to } \Sigma_2(p^{reg}_{abort}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_2(p^{reg}_{abort}(\omega_{qrz})) \end{array} \right\}$$

end

if $\vartheta = imm$ **then**

$$\sigma^{imm}_{abort}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(p^{imm}_{abort}(\omega_{qrz})) \\ \texttt{immediate if } \lambda^b(\sigma^{imm}_{abort}(\omega_{qrz})) \texttt{ abort} \\ \quad \texttt{to } \Sigma_2(p^{imm}_{abort}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_2(p^{imm}_{abort}(\omega_{qrz})) \end{array} \right\}$$

end

▷ initial *and* final *identifiers are optional and are used for the first and last state*

Correctness

To check the correctness of Definition 9.14, the following lemma is used.

Lemma 9.15. *Let* $t^{\Sigma_{abort}}_{qrz \mapsto scc}(p^{\vartheta}_{abort}(\omega_{qrz}))$ *translate* $p^{\vartheta}_{abort}(\omega_{qrz})$ *to* $\sigma^{\vartheta}_{abort}(\omega_{scc'})$ *as specified in Definition 9.14. Then,* $\sigma^{\vartheta}_{abort}(\omega_{scc'})$ *conforms to the syntax rules of* $\sigma^{\vartheta}_{abort}(\omega_{scc})$ *and preserves the SOS transition rules of* $p^{\vartheta}_{abort}(\omega_{qrz})$.

Proof The validity of Lemma 9.15 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma^{\vartheta}_{abort}(\omega_{scc'})$ with the syntax rules of $\sigma^{\vartheta}_{abort}(\omega_{scc})$ as specified in Section 3.5). Second, the semantic correctness is checked by comparing the SOS transition rules of $\sigma^{\vartheta}_{abort}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\sigma^{\vartheta}_{abort}(\omega_{qrz})$ and $\Sigma_2(p^{\vartheta}_{abort}(\omega_{qrz})) \coloneqq \sigma_{nothing}(\omega_{qrz})$ (see Section 3.3).

Illustrative Examples for Lemma 9.15

As an example, Figure 9.18 shows a resulting regular abort transition from `S1` to `S2`. In contrast, Figure 9.19 shows a resulting immediate abort transition from `S1` to `S2` [WS22].

9.3.14. Conditions

This step covers the translation function for translating condition patterns in *Quartz* $P^{\vartheta}_{cond}(\omega_{qrz})$ to condition variants in SCCharts $\Sigma^{\vartheta}_{cond}(\omega_{scc'})$.

```
...
  initial state S1
  if a abort to S2
  final state S2
...
```

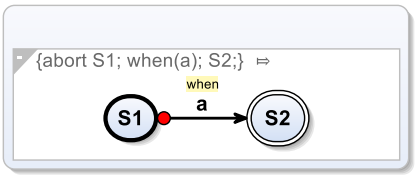


Figure 9.18.: *SCChart of* Quartz *sequence* {`abort S1; when(a); S2;`} *[WS22]*

```
...
  initial state S1
  immediate if a abort to S2
  final state S2
...
```

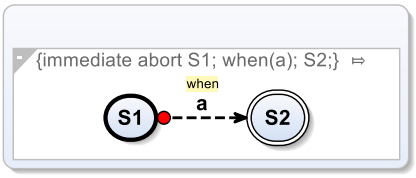


Figure 9.19.: *SCChart of* Quartz *sequence* {`immediate abort S1; when(a); S2;`} *[WS22]*

Definition 9.15 (Translation of conditions – Quartz-to-SCChart). *A condition in* Quartz *has one of the following patterns* $p^{\vartheta}_{cond}(\omega_{qrz}) \in P^{\vartheta}_{cond}(\omega_{qrz})$*:*

- ***if condition*** $\vartheta = it$***)***

$$\Sigma_1(\sigma^{it}_{cond}(\omega_{qrz}))\ \ if(\lambda^b(\sigma^{it}_{cond}(\omega_{qrz})))\{\ \Sigma_2(\sigma^{it}_{cond}(\omega_{qrz}))\ \}\ \Sigma_3(\sigma^{it}_{cond}(\omega_{qrz}))$$

- ***if-else condition*** $\vartheta = ite$***)***

$$\Sigma_1(\sigma^{ite}_{cond}(\omega_{qrz}))\ \ if(\lambda^b(\sigma^{ite}_{cond}(\omega_{qrz})))\{\ \Sigma_2(\sigma^{ite}_{cond}(\omega_{qrz}))\ \}else\{\ \Sigma_3(\sigma^{ite}_{cond}(\omega_{qrz}))\ \}\ \Sigma_4(\sigma^{ite}_{cond}(\omega_{qrz}))$$

Each of these patterns is translated to a corresponding condition in SCCharts $\sigma^{\vartheta}_{cond}(\omega_{scc'}) \in \Sigma^{\vartheta}_{cond}(\omega_{scc'})$ *using the translation function* $t^{\Sigma_{cond}}_{qrz\mapsto scc}(p^{\vartheta}_{cond}(\omega_{qrz}))$*, which is described by Algorithm 35.*

Correctness

To check the correctness of Definition 9.15, the following lemma is used.

Lemma 9.16. $t^{\Sigma_{cond}}_{qrz\mapsto scc}(p^{\vartheta}_{cond}(\omega_{qrz}))$ *translates* $p^{\vartheta}_{cond}(\omega_{qrz})$ *to* $\sigma^{\vartheta}_{cond}(\omega_{scc'})$ *as specified in Definition 9.15.* $\sigma^{\vartheta}_{cond}(\omega_{scc'})$ *conforms to the syntax rules of* $\sigma^{\vartheta}_{cond}(\omega_{scc})$ *and preserves the SOS transition rules of* $p^{\vartheta}_{cond}(\omega_{qrz})$*.*

Proof The validity of Lemma 9.16 is proved as follows: First, the syntactic correctness is checked by comparing the resulting syntax of $\sigma^{\vartheta}_{cond}(\omega_{scc'})$ with the syntax rules of $\sigma^{\vartheta}_{cond}(\omega_{scc})$ as specified in Section 3.5. Second, the semantic correctness is checked by comparing the SOS transition rules of

Algorithm 35 Translate condition – Quartz-to-SCChart

Input: $p^{\vartheta}_{cond}(\omega_{qrz})$
Output: $\sigma^{\vartheta}_{cond}(\omega_{scc'})$
Translation Function $t^{\Sigma_{cond}}_{qrz \mapsto scc}(p^{\vartheta}_{cond}(\omega_{qrz}))$**:**

if ϑ = *it* **then**

$$\sigma^{it}_{cond}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{it}_{cond}(\omega_{qrz})) \\ \texttt{immediate if } \lambda^b(\sigma^{it}_{cond}(\omega_{qrz})) \texttt{ go} \\ \quad \texttt{to } \Sigma_2(\sigma^{it}_{cond}(\omega_{qrz})) \\ \texttt{immediate if !(}\lambda^b(\sigma^{it}_{cond}(\omega_{qrz}))\texttt{) go} \\ \quad \texttt{to } \Sigma_3(\sigma^{it}_{cond}(\omega_{qrz})) \\ \texttt{state } \Sigma_2(\sigma^{it}_{cond}(\omega_{qrz})) \\ \texttt{immediate go to } \Sigma_3(\sigma^{it}_{cond}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_3(\sigma^{it}_{cond}(\omega_{qrz})) \end{array} \right\}$$

end

if ϑ = *del* **then**

$$\sigma^{ite}_{cond}(\omega_{scc'}) \leftarrow \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{immediate if } \lambda^b(\sigma^{ite}_{cond}(\omega_{qrz})) \texttt{ go} \\ \quad \texttt{to } \Sigma_2(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{immediate if !(}\lambda^b(\sigma^{ite}_{cond}(\omega_{qrz}))\texttt{) go} \\ \quad \texttt{to } \Sigma_3(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{state } \Sigma_2(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{immediate go to } \Sigma_4(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{state } \Sigma_3(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{immediate go to } \Sigma_4(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{[final] state } \Sigma_4(\sigma^{ite}_{cond}(\omega_{qrz})) \end{array} \right\}$$

end

▷ initial *and* final *identifiers are optional and are used for the first and last state*

$\sigma^{it}_{cond}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\Sigma_1(\sigma^{it}_{cond}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$, $\sigma^{it}_{cond}(\omega_{qrz})$, and $\Sigma_3(\sigma^{it}_{cond}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$ for pattern 1 (see Section 3.3), and by comparing the SOS transition rules of $\sigma^{ite}_{cond}(\omega_{scc})$ (see Section 3.5) with those of the sequence $\Sigma_1(\sigma^{ite}_{cond}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$, $\sigma^{ite}_{cond}(\omega_{qrz})$, and $\Sigma_4(\sigma^{ite}_{cond}(\omega_{qrz})) := \sigma_{nothing}(\omega_{qrz})$ for pattern 2 (see Section 3.3).

Illustrative Examples for Lemma 9.16

As an example, Figure 9.20 shows an if condition where state `S2` is executed if condition `a` is `true`. Figure 9.21 shows an if-else condition, where state `S2` is executed if condition `a` is `true`, and state `S3` is executed otherwise.

9.3.15. Sequences

This step covers the translation function for translating sequences in *Quartz* models $\Sigma_{seq}(\omega_{qrz})$ and *Quartz* patterns $P(\omega_{qrz})$ to sequences in SCCharts $\Sigma_{seq}(\omega_{scc'})$.

```
...
  initial state S1
  immediate if a go to S2
  immediate if !a go to S3
  state S2
  immediate go to S4
  state S3
  immediate go to S4
  final state S4
...
```

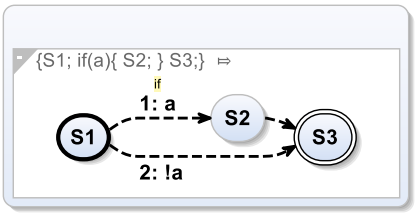


Figure 9.20.: *SCChart of* Quartz *sequence* {`S1; if(a) S2; S3;`} *[WS24a]*

```
...
initial state S1{
  initial state S1
  ...
  final state S3 ""
  immediate go to S1
}
...
```

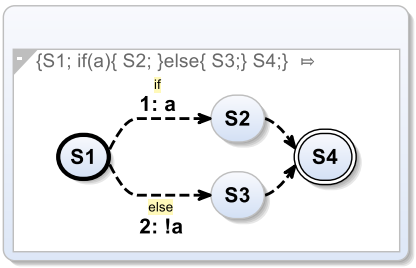


Figure 9.21.: *SCChart of* Quartz *sequence* {`S1; if(a) S2; else S3; S4;`} *[WS22]*

Definition 9.16 (Translation of sequences – Quartz-to-SCChart). *Let* $\Sigma_{seq}(\omega_{qrz})$ *be a set of* Quartz *statements and* $P(\omega_{qrz})$ *a set of* Quartz *patterns that are translated to a set of SCChart statements* $\Sigma_{seq}(\omega_{scc'})$ *according to the Definition 9.7, 9.8, 9.9, 9.10, 9.11, 9.12, 9.13, 9.14, and 9.15. The translation function* $t^{\Sigma_{seq}}_{qrz \mapsto scc}(\Sigma_{seq}(\omega_{scc'}))$ *inserts* $\sigma_i(\omega_{scc'}) \in \Sigma_{seq}(\omega_{scc'})$ *to* $\omega_{scc'}$*, following the process described by Algorithm 36, where* $i \geq 0$*.*

Algorithm 36 Add sequence – Quartz-to-SCChart

Input: $\Sigma_{seq}(\omega_{scc'})$
Output: $\omega_{scc'}$
Translation Function $t^{\Sigma_{seq}}_{qrz \mapsto scc}(\Sigma_{seq}(\omega_{scc'}))$**:**

forall $\sigma_i \in \Sigma_{seq}(\omega_{scc'})$ **do**
  if $i > 0$ **then**
    $\omega_{scc'} \leftarrow \left\{ \begin{array}{l} \texttt{immediate [go|join] to } \sigma_i \\ \texttt{[final] state } \sigma_i \end{array} \right\}$
    ▷ *add* σ_i *to the position w.r.t. its execution order, use* join *for states that contain states (otherwise* go*), and add* final *identifier in last state*
  else
    $\omega_{scc'} \leftarrow \{ \texttt{initial state } \sigma_i \}$
  end
end

Correctness

To check the correctness of Definition 9.16, the following lemma is used.

Lemma 9.17. *Let $\Sigma_{seq}(\omega_{scc'})$ be syntactically correct. Then, $t_{qrz \mapsto scc}^{\Sigma_{seq}}(\Sigma_{seq}(\omega_{scc'}))$ inserts each translated statement $\sigma_i \in \Sigma_{seq}(\omega_{scc'})$ to $\omega_{scc'}$. The resulting SCChart $\omega_{scc'}$ conforms to the syntax rules of ω_{scc} and preserves the SOS transition rules of ω_{qrz}.*

Proof The validity of Lemma 9.17 is proved as follows: First, syntactic correctness is checked by comparing the resulting syntax of $\Sigma_{seq}(\omega_{scc'})$ with the syntax rules of $\Sigma_{seq}(\omega_{scc})$ as specified in Section 3.5 using induction on the number of statements to be added $\Sigma_{seq}(\omega_{scc'})$:

1. **Base Case:** When $\Sigma_{seq}(\omega_{scc'}) = \varnothing$, there are no statements to be added, which conforms to the syntax rules of $\delta_{\omega}(\omega_{scc})$.
2. **Induction Hypothesis:** The lemma holds for any set of statements to be added $\Sigma_{seq}(\omega_{scc'})$.
3. **Inductive Step:** Adding a statement results in a statement to be added that conforms to the syntax rules $\Sigma_{seq}(\omega_{scc})$ in both scenarios $i = 0$ and $i > 0$, because the syntactic correctness of this statement to be added has been proven in the appropriate section.

Second, the overall semantic correctness is given by the individual semantic correctness of each inserted statement.

Illustrative Examples for Lemma 9.17

As an example, Figure 9.22 shows a sequence of inserted statements `S1`, `S2`, `S3`, and `Sn`. Switching between statements does not consume time, and if the individual statements terminate instantaneously, the sequence also terminates instantaneously [WS22]. In addition, switching of sequences containing other sequences (such as `S2`) is indicated by the green triangle [WS22].

```
1  ...
2    initial state S1
3    immediate go to S2
4    state S2 { ... }
5    immediate join to S3
6    state S3
7    immediate go to Sn
8    final state Sn
9  ...
```

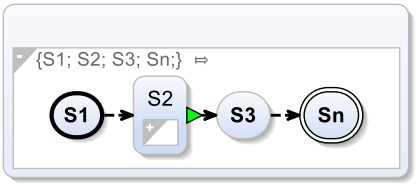


Figure 9.22.: *SCChart of* Quartz *sequence* {`S1; S2; S3; Sn;`} *[WS22]*

9.4. SCChart Optimization

This section defines optimization strategies $T_{scc \mapsto scc}(\omega_{scc})$ for a resulting Quartz-based SCChart $\omega_{scc'} \in \Omega_{scc}$ that are derived from related work in the context of synthesized safe state machines from *Esterel* [PTH06].

9.4.1. Flattening Hierarchy

This step covers the strategy for flattening the state hierarchy, where the focus is on removing surrounding states, which contain a single initial state that optionally followed by other states [PTH06].

Definition 9.17 (Optimization of hierarchy – Quartz-to-SCChart). *A surrounding state in Quartz-based SCCharts, which contains a single initial state that optionally followed by other states has one of the following patterns:*

- ***Pattern 1: Initial state only***

```
initial state Sx{ initial state Sy{ ... } }
```

- ***Pattern 2: Initial state with one other state***

```
initial state Sx{
  initial state Sy
  ... to Sz
  [final] state Sz{ ... } }
```

In both scenarios, the surrounding state is removed as follows, using a translation denoted as $t_{scc \mapsto scc}^{\Sigma_{hierarchy}}(\Sigma(\omega_{scc}))$*:*

- ***Pattern 1: Initial state only***

```
initial state Sy "Sx + Sy"{ ... }
```

- ***Pattern 2: Initial state with one other state***

```
initial state Sy "Sy + Sx"
  ... to Sz
  [final] state Sz{ ... } }
```

Correctness

To check the correctness of Definition 9.17, the following lemma is used.

Lemma 9.18. *Removing the surrounding state following Definition 9.17 does not change the conditions for the termination or transition behavior of the Quartz-based SCChart.*

Proof Lemma 9.18 is valid, because the transition from an initial state to another state contains only the immediate transition, which executes the *nothing* statement. If only the initial state is available, no transition is executed. Additionally, this is confirmed by manually comparing the synthesized *Sequentially Constructive Graphs* [Han+13] of both models (with and without hierarchy optimization) using *KIELER*, e.g., for the following illustrative example, where additional temporary variables that are immediately assigned are ignored.

Illustrative Example for Lemma 9.18

As an example, Figure 9.23 illustrates pattern 2 before optimization and Figure 9.24 illustrates pattern 2 after optimization, where the surrounding state `Sx` is removed.

```
1 ...
2 initial state Sx{
3   initial state Sy
4   immediate abort to Sz
5   final state Sz
6 }
7 ...
```

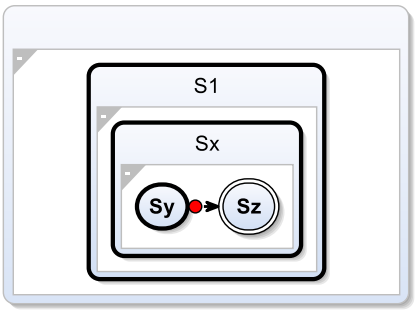


Figure 9.23.: *SCChart before hierarchy optimization (Pattern 2)*

```
1 ...
2   initial state Sy "Sy + Sx"
3   immediate abort to Sz
4   final state Sz
5 ...
```

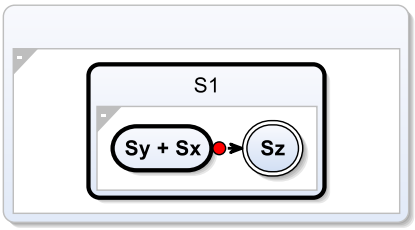


Figure 9.24.: *SCChart after hierarchy optimization (Pattern 2)*

9.4.2. Removing States

This step covers the strategy for removing states, where the focus is on states without internal actions [PTH06].

Definition 9.18 (Optimization of states – Quartz-to-SCChart). *A state in Quartz-based SCCharts can be removed under the following conditions using the translation function denoted as* $t_{scc \to scc}^{\Sigma_{state}}(\Sigma(\omega_{scc}))$*:*

- *The state is an initial state, has no internal actions, and has only one outgoing immediate transition without condition and action.*
- *The state has no internal actions and has only one outgoing immediate transition without condition and action. The incoming transition of the subsequent state is redirected accordingly.*

- *The state has no internal actions, only one incoming immediate transition without condition, and one outgoing transition without condition. The incoming transition of the subsequent state is redirected accordingly, and actions are combined.*
- *The state is a final state, has no internal actions, and has no outgoing transitions.*

Correctness

To check the correctness of Definition 9.18, the following lemma is used.

Lemma 9.19. *Removing states following Definition 9.18 does not change the conditions for the termination or transition behavior of the Quartz-based SCChart.*

Proof Lemma 9.19 is valid, because the considered states contain only immediate outgoing transitions (which execute the *nothing* statement) or no outgoing transitions. Additionally, this is confirmed by manually comparing the synthesized *Sequentially Constructive Graph* of both models (with and without state optimization) using *KIELER*, e.g., for the following illustrative example, where additional temporary variables that are immediately assigned are ignored.

Illustrative Example for Lemma 9.19

As an example, Figure 9.25 illustrates all scenarios before optimization and Figure 9.26 after optimization.

```
1  ...
2    initial state Sa
3    immediate go to Sb
4    state Sb{ ... }
5    join to Sc
6    state Sc
7    immediate do f = true go
         to Sd
8    state Sd{ ... }
9    join to Se
10   final state Se
11 ...
```

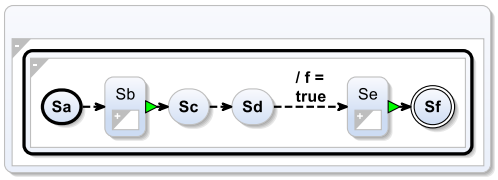


Figure 9.25.: *SCChart before state optimization*

9.5. Experimental Results

The applicability of the introduced *Quartz* code refactoring and transformation to control-flow oriented SCCharts is evaluated with the *Quartz* models

```
1  ...
2    initial state Sb{ ... }
3    do f = true join to Sd
4    final state Sd{ ... }
5  ...
```

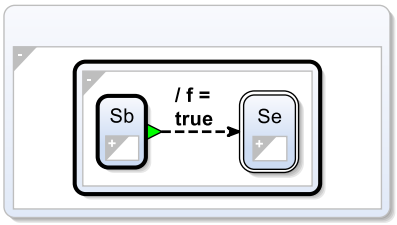


Figure 9.26.: *SCChart after state optimization*

listed in Table 9.1. For evaluation purposes, the expected SCCharts are manually implemented in *KIELER* and the refactoring and translation functions as a prototype in *PLCreX*, resulting in the overall test strategy shown in Figure 9.27. Similar to the previous chapters, the correctness of the automatically generated control-flow oriented SCCharts is verified in two ways: (1) manual reviews are used to identify differences between the expected SCCharts and the automatically generated SCCharts, and (2) the built-in compilers of *KIELER* and *Averest* are used to verify the syntactic correctness of the refactored and automatically generated models.

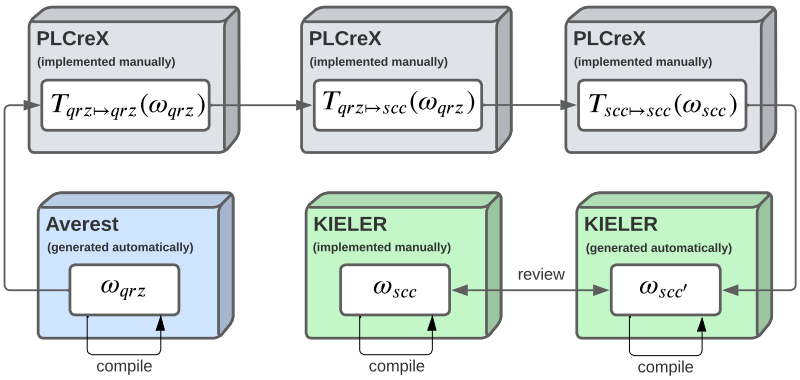


Figure 9.27.: *Test strategy to evaluate the Quartz-to-SCChart transformation including optimization and* Quartz *code refactoring*

Both tests passed for all examples (ignoring minor formatting differences between manually implemented and automatically generated SCCharts), with the following warnings:

- **Supported operators**: According to Lemma 9.6, only a subset of the considered *Quartz* expressions can be translated to corresponding internal SCChart expressions. Therefore, the example *arithmetic operators* throws a warning for affected expression. Affected expressions are skipped during translation.
- **Supported data types**: According to Lemma 9.5, only a subset of the considered *Quartz* data types can be translated to corresponding SCChart data types. Therefore, the example *data types and fields* throws a warning for affected declarations. Affected declarations and expressions

Table 9.1.: *Set of examples and test results to evaluate the applicability of the introduced Quartz-to-SCChart transformation*

Model	Source	ω_{qrz}	ω_{scc}	Result
Alarm function	[Sch19]	C.2	H.1	passed
Foot-controlled loop	self	C.13	H.2	passed
Head-controlled loop	self	C.14	H.3	passed
Arithmetic operators	self	C.5	H.4	passed with warnings
RS-Flip-Flop	[GDV14]	C.23	H.5	passed
2-of-3 logic function	[Sch19]	C.1	H.6	passed
Boolean operators	self	F.7	H.7	passed
Data types and fields	self	F.9	H.8	passed with warnings
KV Diagram optimized Chart	[Kar18]	F.12	H.9	passed
Left detection	[Sch19]	F.13	H.10	passed
Roll Down Shutters	[AG20]	F.17	H.11	passed
Thermometer Code System	[Bub17]	F.27	H.12	passed
Toggle Switch 4x	[Bub17]	F.28	H.13	passed

are skipped during translation.

Based on the experimental results in Table 9.1, it can be concluded that the introduced refactoring and translation functions are applicable and lead to correct SCCharts intended for an alternative model view. As a full example, Figure 9.28 demonstrates the different steps from refactoring the initial *Quartz* model (see Figure 9.28a and Figure 9.28a), to the transformation into a control-flow oriented SCChart with subsequent optimization (see Figure 9.28d) using an FBD-based *Quartz* example introduced in [WS22][7].

9.6. Summary

This chapter introduced a *Quartz* code refactoring and transformation to control-flow oriented SCCharts. For this purpose, a code refactoring, individual translation functions, and optimizations were defined, whose applicability was demonstrated using a set of *Quartz* examples. Based on the presented lemmas and experimental results, the following theorem encapsulates the entire transformation:

Theorem 9.1 (Quartz-to-SCChart Translation). *Let $T_{qrz\mapsto qrz}(\omega_{qrz})$ be the code refactoring of ω_{qrz}, $T_{qrz\mapsto scc}(\omega_{qrz})$ be the model transformation of ω_{qrz} to $\omega_{scc'}$, and $T_{scc\mapsto scc}(\omega_{scc'})$ be the model optimization of $\omega_{scc'}$ as introduced in this chapter. Then, the resulting control-flow oriented SCChart $\omega_{scc'}$:*

[7] Note that `S0` is empty, which is why it does not appear in the resulting SCChart.

1. *Conforms to the syntax rules of* ω_{scc}
2. *Preserves the semantics of* ω_{qrz}
3. *Contains constructs corresponding to the constructs of* ω_{qrz} *and preserves the intended functionality of* ω_{qrz} *under the following conditions:*
   - *no imported* Quartz *models*
   - $\Delta_{idcl}(\omega_{qrz}) = \Delta_{in}(\omega_{qrz}) \cup \Delta_{out}(\omega_{qrz}) \cup \Delta_{inout}(\omega_{qrz})$
   - $\Delta_{vdcl}(\omega_{qrz}) = \Delta_{local}(\omega_{qrz})$
   - $\forall \alpha^{[+]}(\omega_{qrz}) : \alpha^{[+]} \in \{\alpha_{bv}^{bool}, \alpha_{i}^{int}, \alpha_{i}^{dint}, \alpha_{i}^{uint}, \alpha_{dur}, \alpha^{+}\}$, *where* α_{dur} *can be treated as an bounded integer and is specified in milliseconds*
   - $\forall \tau(\omega_{qrz}) : \tau \in \{\texttt{emit}(\pi), \tau_{misc}^{cst}, \tau_{misc}^{id}, \tau_{misc}^{\pi,\eta,\lambda}, \tau_{misc}^{br}, \tau_{misc}^{true}, \tau_{misc}^{false}, \tau_{misc}^{arr}, \tau_{misc}^{inv}, \tau_{comp}^{eq}, \tau_{comp}^{ne}, \tau_{comp}^{gt}, \tau_{comp}^{ge}, \tau_{comp}^{lt}, \tau_{comp}^{le}, \tau_{arith}^{mul}, \tau_{arith}^{div}, \tau_{arith}^{add}, \tau_{arith}^{sub}, \tau_{arith}^{expt}, \tau_{arith}^{mod}, \tau_{arith}^{um}, \tau_{cond}^{sel}\}$
   - $\forall \sigma(\omega_{qrz}) : \sigma \in \{\sigma_{nothing}, \sigma_{await}^{\vartheta}, \sigma_{pause}, \sigma_{conc}, \sigma_{loop}^{\vartheta}, \sigma_{halt}, \sigma_{abort}^{\vartheta}, \sigma_{cond}^{\vartheta}, \sigma_{ass}^{\vartheta}\}$

Proof The validity of Theorem 9.1 is proved as follows:

1. **Syntax Conformance**: Lemma 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9, 9.10, 9.11, 9.12, 9.13, 9.14, 9.15, 9.16, and 9.17 demonstrate that each translated construct conforms to the syntax rules of ω_{scc} as specified in Section 3.5. Furthermore, Lemma 9.18 and 9.19 demonstrate that the optimized SCChart conforms to the syntax rules of ω_{scc} as specified in Section 3.5.
2. **Semantic Preservation**: The following lemmas address the preservation of semantics for their respective constructs.
   - Code refactoring: Lemma 9.1
   - Model declaration: Lemma 9.2
   - Interfaces: Lemma 9.3
   - Variables: Lemma 9.4
   - Data types and fields: Lemma 9.5
   - Expressions: Lemma 9.6
   - Immediate transition: Lemma 9.7
   - Await: Lemma 9.8
   - Pause: Lemma 9.9
   - Assignments: Lemma 9.10 and 9.11

- Synchronous Concurrency: Lemma 9.12
- Loop: Lemma 9.13
- Halt: Lemma 9.14
- Abort: Lemma 9.15
- Conditions: Lemma 9.16
- Sequences: Lemma 9.17

3. **Construct Correspondence**: Given the conditions of the theorem, the provided definitions, proofs, and experimental results, it can be concluded that the translation functions produce corresponding constructs in ω_{scc} for the considered constructs in ω_{qrz}, preserving the original functionality, taking into account the previous refactoring and subsequent optimizations.

Overall, this results in the following solutions to the challenges summarized in Section 9.1.

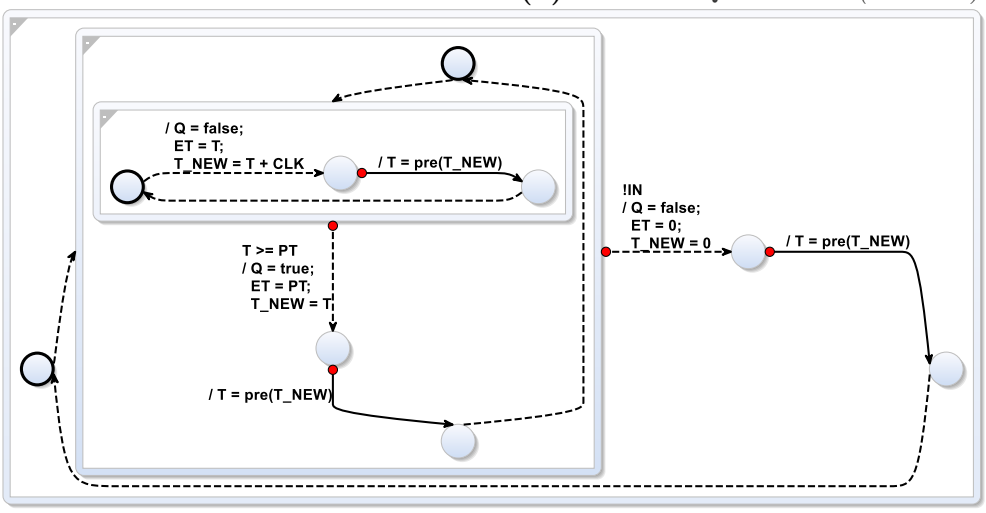
1. **Correct code refactoring of the initial *Quartz* model:** The solution follows from Section 9.2.
2. **Correct and complete translation of the considered *Quartz* constructs:** The solution follows from Section 9.3.
3. **Correct optimization of the resulting SCChart:** The solution follows from Section 9.4.

```
loop{
  if(IN){                  // c2
    if(T >= PT){           // c1
      Q = true;            // S1
      ET = PT;             // S1
      T_NEW = T;           // S1
    }else{
      Q = false;           // S2
      ET = T;              // S2
      T_NEW = T + CLK;     // S2
    }
  }else{
    Q = false;             // S4
    ET = 0;                // S4
    T_NEW = 0;             // S4
    }
    next(T) = T_NEW;       // S3
    pause;                 // S3
}
```

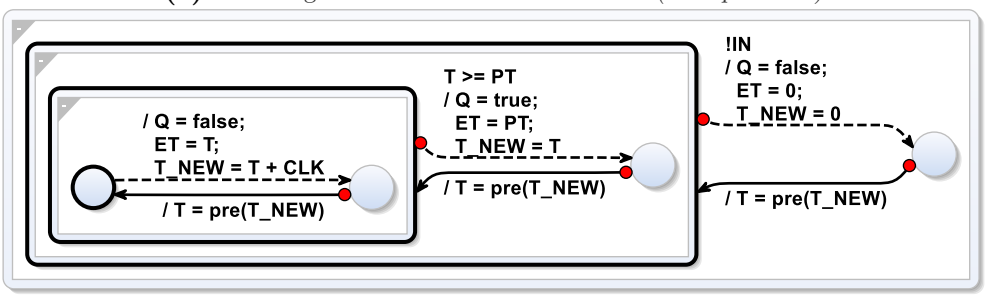
(a) *Initial Quartz Model*

```
loop{
  immediate abort{
    loop{
      immediate abort{
        loop{
          Q = false;           // S2
          ET = T;              // S2
          T_NEW = T + CLK;     // S2
          next(T) = T_NEW;     // S3
          pause;               // S3
        }
      }when(T >= PT);          // c1
      Q = true;                // S1
      ET = PT;                 // S1
      T_NEW = T;               // S1
      next(T) = T_NEW;         // S3
      pause;                   // S3
    }
  }when(!IN);                  // !c2
  Q = false;                   // S4
  ET = 0;                      // S4
  T_NEW = 0;                   // S4
  next(T) = T_NEW;             // S3
  pause;                       // S3
}
```

(b) *Refactored Quartz Model (Pattern 3)*



(c) *Resulting control-flow oriented SCChart (not optimized)*



(d) *Resulting control-flow oriented SCChart (optimized)*

Figure 9.28.: *From Quartz to control-flow oriented SCChart [WS22]*

Chapter 10

Conclusions

This thesis explored the reuse of existing IEC-61131-3 ST- and FBD-based POUs in model-based design (which supports formal verification) by translating them into synchronous models. The central objective was to investigate, using *Quartz*, SCL and SCCharts as examples, whether ST- and FBD-based POUs can be translated into synchronous models while maintaining their behavioral semantics. In this context, it has been shown that the structural complexity of data-flow models in real-world applications can be reduced using formal methods and formula refactoring. Additionally, the possibility of translating ST- and FBD-based *Quartz* models into control-flow oriented SCCharts has been presented. In this context, specific patterns have been identified that allow the transformation of FBD-based *Quartz* models into hierarchical control-flow oriented SCCharts. A summary of the contributions is shown in Figure 10.1, where the elements are labeled according to the categories below. These categories contain the key findings and answers to the hypotheses **H1**, **H2** and **H3** that were formulated in Section 1.1.

1. **H1: Model-Based Design of Program Organization Units**

   A detailed model transformation from textual ST models to corresponding *Quartz* and SCL models, and from graphical FBDs to data-flow oriented *Quartz* models and SCCharts were presented. The correctness of the transformations has been demonstrated by theoretical reasoning, and the theoretical results have been evaluated with real-world and self-defined IEC 61131-3 examples, with the translation functions implemented as a prototype in *PLCreX*. As shown in Theorem 4.1, 5.1, 6.1, and 7.1, the transformations preserve the original runtime behavior of ST- and FBD-based POUs. Furthermore, it has been shown that existing variable and instance names are retained, and additional variables provide traceability to the original IEC 61131-3 POU and internal ports. Existing control structures such as loops and conditions are also preserved. This allows an intuitive post-translation modification. Overall, this translation enables reuse in model-based design using *KIELER* and *Averest*, which support formal verification.

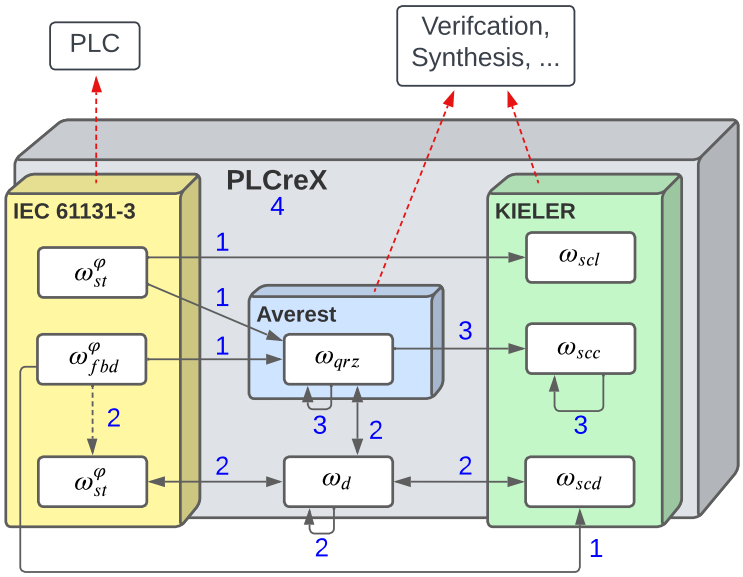


Figure 10.1.: *Contribution Summary*

2. **H2: Formal Methods-Based Optimization of Data-Flow Models**

   By introducing a configurable optimization strategy based on *NuSMV*, SMT *Z3Py*, and formula refactoring, a formal methods-based optimization process for data-flow models has been developed. As shown in Theorem 8.1, the optimization approach effectively detects and minimizes the structural complexity of data-flow models in real-world applications without changing the logical behavior of the models, where structural complexity is measured by the number of variable accesses (including output variables of instances), operators, and edges. Empirical evaluations using industrial examples from PLC vendors and literature demonstrated significant optimization potential, with the optimization process implemented in *PLCreX*, confirming that data-flow models in real-world applications are frequently presented in a more complex structure than their logic requires.

3. **H3: Control-Flow Oriented SCCharts of *Quartz* Models**

   A detailed model transformation from ST- and FBD-based *Quartz* models to control-flow oriented SCCharts has been presented. This transformation takes into account hierarchy and state optimization of the resulting SCCharts. The correctness of the translation functions has been demonstrated by theoretical reasoning, and the theoretical results have been evaluated with ST- and FBD-based *Quartz* models, with the translation functions and optimizations implemented as a prototype in *PLCreX*. As shown in Theorem 9.1, by detecting specific patterns, *Quartz* models can be translated into hierarchical control-flow oriented SCCharts, providing an alternative control flow view for system analysis

and design.

4. ***PLCreX***

   PLCreX has been developed as part of this thesis. It's a project for simplification, transformation, analysis, and validation of IEC 61131-3 POUs. *PLCreX* encapsulates the approaches developed in this thesis and serves as a tool to support in challenges of real-world applications.

While this thesis offers robust translations and specific tool support, additional research and development would further enhance the applicability. For example, adding more language features to SCL (such as importing other models or foot-controlled loops) or more detailed documentation, or considering external data types and operators in the introduced approaches could make the transformation process more intuitive for professionals with an industrial background. Furthermore, variables in resulting SCL models that are derived from POUs without memory are reset by default, even if they are updated within the same *macro step*. Resetting is not necessary in these cases. In addition, initializing variables in *Quartz* at the time of declaration, rather than through separate immediate assignments, would reduce overall lines of code and potentially simplify the traceability with regard to the original IEC 61131-3 model.

Additional research is recommended for translating FBDs to data-flow oriented SCCharts and *Quartz* models. The approaches presented in this thesis enforce the preservation of sequential execution in the original FBDs including instances, resulting in statements being executed sequentially although they could be executed in parallel. The decomposition of POUs into independent paths is recommended.

Furthermore, when translating FBDs to data-flow oriented SCCharts following the approach of additional `MOVE` blocks, there are intermediate values at the input ports of these `MOVE` blocks when at least one instance in the model does not terminate within the current *macro step* (or loop iteration, respectively), which may be the result of invalid operations. These results do not affect the original semantics (and do not cause a runtime error in C-based simulations in *KIELER*) because they are not passed until the `MOVE` blocks have been triggered accordingly. However, the affected expressions should be protected from invalid operations.

Additionally, future research could explore more complex refactoring patterns within the synchronous paradigm to alternatively visualize the models and potentially identify additional hierarchical structures in data-flow or control-flow oriented models. Investigating alternative forms of hierarchical decomposition or synchronous concurrency could extend the approaches presented in this thesis. For example, LLMs could be used to identify similar patterns or control structures in large applications, or to suggest appropriate refactoring strategies in individual POUs.

Furthermore, the formal methods-based optimization introduced in this thesis employs the *Z3Py* theorem solver and the *NuSMV* model checker. Other model checkers, theorem solvers, algorithms, or minimization techniques (with

and without LLM support) could increase the observed minimization potential. Further research could investigate the impact of the introduced optimization strategies on code size and execution speed across different platforms. In addition, it would be beneficial if the optimization approach could take into account user-defined patterns or templates, which are treated as non-modifiable and thus excluded from optimization.

Moreover, *PLCreX* would benefit from additional IEC 61131-3 features and language constructs to apply the introduced methods to more complex real-world applications. Since *PLCreX* is based on an ST-like intermediate representation, other languages can be easily integrated.

In summary, this thesis presents detailed approaches for transforming existing ST- and FBD-based POUs into synchronous models, an optimization strategy for data-flow oriented models, and the synthesis of control-flow oriented SCCharts from ST- and FBD-based *Quartz* models. This enables the transition from traditional IEC 61131-3 development to model-based design using synchronous languages, while reusing existing IEC 61131-3 development efforts.

Bibliography

[AG20] Siemens AG. *LOGO!-Library – Simple application examples - ID: 109783504 - Industry Support Siemens*. `https://support.industry.siemens.com/cs/document/109783504/`. Dec. 2020.

[And95] Charles André. "Syncharts: A visual representation of reactive behaviors". In: Université de Nice-Sophia Antipolis, 1995.

[BAV08] Gülden Bayrak, Farisoroosh Abrishamchian, and Birgit Vogel-Heuser. "Effiziente Steuerungsprogrammierung durch automatische Modelltransformation von Matlab/Simulink/Stateflow nach IEC 61131-3". In: *Automatisierungstechnische Praxis (atp)* 50.12 (2008), pp. 49–55.

[BB13] Jan Olaf Blech and Sidi Ould Biha. "On Formal Reasoning on the Semantics of PLC using Coq". In: *ArXiv* (2013), pp. 1–35. eprint: `arXiv:1301.3047v1`.

[BB91] Albert Benveniste and Gérard Berry. "The synchronous approach to reactive and real-time systems". In: *Proceedings of the IEEE*. Vol. 79. 9. 1991, pp. 1270–1282. DOI: `10.1109/5.97297`.

[BBK12] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. "Arcade.PLC: A verification platform for programmable logic controllers". In: *27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*. 2012, pp. 338–341. ISBN: 9781450312042. DOI: `10.1145/2351676.2351741`.

[BD12] Haniel Barbosa and David Déharbe. "Formal Verification of PLC Programs Using the B Method". In: *Abstract State Machines, Alloy, B, VDM, and Z*. Ed. by John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 353–356. ISBN: 978-3-642-30885-7.

[BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. "A Tutorial on Uppaal". In: *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*. Ed. by Marco Bernardo and Flavio Corradini. Berlin, Heidelberg:

Springer Berlin Heidelberg, 2004, pp. 200–236. ISBN: 978-3-540-30080-9. DOI: 10.1007/978-3-540-30080-9_7. URL: https://doi.org/10.1007/978-3-540-30080-9_7.

[Bec+15] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. *Regression Verification for Programmable Logic Controller Software*. Tech. rep. 6. Karlsruher Institut für Technologie (KIT), 2015. 16 pp. DOI: 10.5445/IR/1000047251.

[Ben+03] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert Simone. "The Synchronous Languages 12 Years Later". In: *Proceedings of the IEEE* 91 (Feb. 2003), pp. 64–83. DOI: 10.1109/JPROC.2002.805826.

[Ber16] Yves Bertot. "Coq in a Hurry". 3rd cycle. Lecture. Types Summer School, also used at the University of Goteborg, Nice, Ecole Jeunes Chercheurs en Programmation, Universite de Nice, France, June 2016. URL: https://cel.hal.science/inria-00001173.

[BG92] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(92)90005-V.

[Bia16] Sebastian Biallas. *Verification of Programmable Logic Controller Code using Model Checking and Static Analysis*. Aachen: Shaker Verlag GmbH, 2016. ISBN: 978-3-8440-4711-0.

[BM14] Sonam Bhatia and Jyoteesh Malhotra. "A survey on impact of lines of code on software complexity". In: *2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014)*. Aug. 2014, pp. 1–4. DOI: 10.1109/ICAETR.2014.7012875.

[Bub17] Otto Bubbers. *Lösungen: Skript Steuerungstechnik für UT*. Gottlieb-Daimler-Schule 2, Technisches Schulzentrum Sindelfingen mit Abteilung Akademie für Datenverarbeitung. Sindelfingen, 2017.

[Cav+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 334–342. ISBN: 978-3-319-08867-9.

[CGP01] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Jan. 2001. ISBN: 978-0-262-03270-4.

[Cim+00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. "NUSMV: a new symbolic model checker". In: *STTT* 2 (Mar. 2000), pp. 410–425. DOI: 10.1007/s100090050046.

[CM03] Dominique Cansell and Dominique Mery. "Foundations of the B Method." In: *Computers and Informatics* 22 (Jan. 2003), 31 p.

[CR14] Luis Cruz Salazar and Oscar Rojas Alvarado. "The Future of Industrial Automation and IEC 614993 Standard". In: Oct. 2014. DOI: 10.1109/CIIMA.2014.6983434.

[DBF15] Dániel Darvas, Enrique Blanco Vinuela, and Borja Fernández Adiego. "PLCverif: A Tool to Verify PLC Programs Based on Model Checking Techniques". In: *15th International Conference on Accelerator and Large Experimental Physics Control Systems*. 2015. DOI: 10.18429/JACoW-ICALEPCS2015-WEPGF092.

[DMB16] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. "Generic representation of PLC programming languages for formal verification". In: *Proceedings of the 23rd PhD Mini-Symposium* (Budapest, Hungary). Zenodo, Feb. 2016, pp. 6–9. DOI: 10.5281/zenodo.51064.

[DV12] Wenbin Dai and Valeriy Vyatkin. "Redesign Distributed PLC Control Systems Using IEC 61499 Function Blocks". In: *IEEE Transactions on Automation Science and Engineering* 9.2 (2012), pp. 390–401. DOI: 10.1109/TASE.2012.2188794.

[Ebr+23] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Wasif Afzal, and Cristina Seceleanu. "PyLC: A Framework for Transforming and Validating PLC Software using Python and Pynguin Test Generator". In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. SAC '23. Tallinn, Estonia: Association for Computing Machinery, 2023, pp. 1476–1485. ISBN: 9781450395175. DOI: 10.1145/3555776.3577698. URL: https://doi.org/10.1145/3555776.3577698.

[Eno+16] Eduard Enoiu, Daniel Sundmark, Adnan Causevic, Robert Feldt, and Paul Pettersson. "Mutation-Based Test Generation for PLC Embedded Software Using Model Checking". In: *Testing Software and Systems*. Cham: Springer International Publishing, Oct. 2016, pp. 155–171. ISBN: 978-3-319-47442-7. DOI: 10.1007/978-3-319-47443-4_10.

[Fer+15] Borja Fernandez Adiego, Daniel Darvas, Enrique Blanco Vinuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Victor Manuel Gonzalez Suarez. "Applying Model Checking to Industrial-Sized PLC Programs". In: *IEEE Transactions on Industrial Informatics* 11.6 (2015), pp. 1400–1410. ISSN: 1551-3203. DOI: 10.1109/TII.2015.2489184M4-Citavi.

[GDV14] Electronic German Commission for Electrical, Information Technologies of DIN, and VDE. *Programmable controllers - Part 3: Programming languages (IEC 61131-3:2013); German version EN 61131-3:2013*. Standard. Berlin: DIN German Institute for Standardization, 2014.

[Gom+09] Daniel Gomez-Prado, Qian Ren, Maciej J. Ciesielski, Jérémie Guillot, and Emmanuel Boutillon. "Optimizing data flow graphs to minimize hardware implementation". In: *2009 Design, Automation & Test in Europe Conference & Exhibition* (2009), pp. 117–122.

[Gri+20] Lena Grimm, Steven Smyth, Alexander Schulz-Rosengarten, Reinhard von Hanxleden, and Marc Pouzet. "From Lustre to Graphical Models and SCCharts". In: *2020 Forum for Specification and Design Languages (FDL)*. 2020, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232944.

[Hal+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320. DOI: 10.1109/5.97300.

[Hal98] Nicolas Halbwachs. "Synchronous programming of reactive systems - A tutorial and commented bibliography". In: *LNCS* 1427 (Jan. 1998), pp. 1–16.

[Han+13] Reinhard Hanxleden, Michael Mendler, Joaquin Aquado, Björn Duderstadt, Insa Marie-Ann Fuhrmann, Christian Motika, Stephan Mercer, Owen O'Brien, and Partha Roop. *Sequentially Constructive Concurrency: A Conservative Extension of the Synchronous Model of Computation*. Tech. rep. [Online]. Available: https://nbn-resolving.org/urn:nbn:de:101:1-201402136745. Kiel: Selbstverlag des Instituts für Informatik, 2013.

[Han+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications". In: *SIGPLAN Not.* 49.6 (June 2014), pp. 372–383. ISSN: 0362-1340. DOI: 10.1145/2666356.2594310.

[Har08] John Harrison. "Theorem Proving for Verification (Invited Tutorial)". In: *Computer Aided Verification*. Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 11–18. ISBN: 978-3-540-70545-1.

[Har87] David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: https://doi.org/10.1016/

0167-6423(87)90035-9. URL: https://www.sciencedirect.com/science/article/pii/0167642387900359.

[Hua+19] Yanhong Huang, Xiangxing Bu, Gang Zhu, Xin Ye, Xiaoran Zhu, and Jianqi Shi. "KST: Executable Formal Semantics of IEC 61131-3 Structured Text for Verification". In: *IEEE Access* 7 (2019), pp. 14593–14602. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2894026.

[Jee+09] Eunkyoung Jee, Junbeom Yoo, Sungdeok Cha, and Doohwan Bae. "A data flow-based structural testing technique for FBD programs". In: *Information and Software Technology* 51.7 (2009), pp. 1131–1139. ISSN: 09505849. DOI: 10.1016/j.infsof.2009.01.003.

[JR01] Fernando Jimenez-Fraustro and Eric Rutten. *A synchronous model of IEC 61131 PLC languages in SIGNAL*. 2001. DOI: 10.1109/EMRTS.2001.934016.

[JT10] Karl Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming industrial automation systems: Concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Berlin Heidelberg, 2010, pp. 1–390. ISBN: 9783642120145. DOI: 10.1007/978-3-642-12015-2.

[Kar18] Cihat Karaali. *Grundlagen der Steuerungstechnik*. de. 3rd ed. Wiesbaden, Germany: Springer Fachmedien, May 2018.

[Kar53] Maurice Karnaugh. "The map method for synthesis of combinational logic circuits". In: *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics* 72.5 (1953), pp. 593–599. DOI: 10.1109/TCE.1953.6371932.

[Kas+24] Maximilian Kasperowski, Niklas Rentz, Sören Domrös, and Reinhard von Hanxleden. "KIELER: A Text-First Framework for Automatic Diagramming of Complex Systems". In: Sept. 2024, pp. 402–418. ISBN: 978-3-031-71290-6. DOI: 10.1007/978-3-031-71291-3_33.

[Kel+01] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. *A Practical Tutorial on Modified Condition/Decision Coverage*. Tech. rep. 2001.

[Koz+24] Heiko Koziolek, Virendra Ashiwal, Soumyadip Bandyopadhyay, and Chandrika K R. *Automated Control Logic Test Case Generation using Large Language Models*. 2024. arXiv: 2405.01874 [cs.SE]. URL: https://arxiv.org/abs/2405.01874.

[LB21] Guolin Lyu and Robert William Brennan. "Towards IEC 61499-Based Distributed Intelligent Automation: A Literature Review". In: *IEEE Transactions on Industrial Informatics* 17.4 (2021), pp. 2295–2306. DOI: 10.1109/TII.2020.3016990.

[Le +11] Thierry Le Sergent, Alain Guennec, Sébastien Gérard, Yann Tanguy, and François Terrier. "Using SCADE System for the Design and Integration of Critical Systems". In: Oct. 2011. DOI: 10.4271/2011-01-2577.

[Le +91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. "Programming real-time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1321–1336. DOI: 10.1109/5.97301.

[Lew98] Robert W. Lewis. *Programming Industrial Control Systems Using IEC 1131-3 (I E E Control Engineering Series)*. 1998. ISBN: 0852969503.

[LF22] Stephan Lukasczyk and Gordon Fraser. "Pynguin: automated unit test generation for Python". In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ICSE '22. ACM, May 2022. DOI: 10.1145/3510454.3516829. URL: http://dx.doi.org/10.1145/3510454.3516829.

[Liu+24] Zihan Liu, Ruinan Zeng, Dongxia Wang, Gengyun Peng, Jingyi Wang, Qiang Liu, Peiyu Liu, and Wenhai Wang. *Agents4PLC: Automating Closed-loop PLC Code Generation and Verification in Industrial Control Systems using LLM-based Agents*. 2024. arXiv: 2410.14209 [cs.SE]. URL: https://arxiv.org/abs/2410.14209.

[New+16] Josh Newell, Linna Pang, David Tremaine, Alan Wassyng, and Mark Lawford. "Formal Translation of IEC 61131-3 Function Block Diagrams to PVS with Nuclear Application". In: *NASA Formal Methods.8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*. Ed. by Sanjai Rayadurgam and Oksana Tkachuk. Vol. 9690 SV -. Lecture Notes in Computer Science TS - CrossRef. Cham: Springer International Publishing, 2016, pp. 206–220. ISBN: 978-3-319-40647-3. DOI: 10.1007/978-3-319-40648-0_16M4-Citavi.

[New+18] Josh Newell, Linna Pang, David Tremaine, Alan Wassyng, and Mark Lawford. "Translation of IEC 61131-3 Function Block Diagrams to PVS for Formal Verification with Real-Time Nuclear Application". In: *Journal of Automated Reasoning* 60 (Jan. 2018), pp. 1–22. DOI: 10.1007/s10817-017-9415-7.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. "PVS: A prototype verification system". In: *International Conference on Automated Deduction*. Springer. 1992, pp. 748–752.

[PE10] Olivera Pavlovic and Hans-Dieter Ehrich. "Model Checking PLC Software Written in Function Block Diagram". In: *2010 Third International Conference on Software Testing, Verification and*

Validation. IEEE, 2010, pp. 439–448. ISBN: 9780769539904. DOI: 10.1109/ICST.2010.10.

[PLC09] PLCopen. *Technical Paper PLCopen Technical Committee 6, XML Formats for IEC 61131-3 Version 2.01 – Official Release*. Tech. rep. 2009.

[Plo04] Gordon Plotkin. "A Structural Approach to Operational Semantics". In: *J. Log. Algebr. Program.* 60-61 (July 2004), pp. 17–139. DOI: 10.1016/j.jlap.2004.05.001.

[Pra+17] Herbert Prahofer, Florian Angerer, Rudolf Ramler, and Friedrich Grillenberger. "Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application". In: *IEEE Transactions on Industrial Informatics* 13.1 (2017), pp. 37–47. ISSN: 15513203. DOI: 10.1109/TII.2016.2604760.

[PTH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. "Synthesizing Safe State Machines from Esterel". In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*. LCTES '06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 113–124. ISBN: 159593362X. DOI: 10.1145/1134650.1134667.

[Rob10] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.6 User Manual*. Povo (Trento) – Italy, 2010.

[Rös+15] Susanne Rösch, Sebastian Ulewicz, Julien Provost, and Birgit Vogel-Heuser. "Review of Model-Based Testing Approaches in Production Automation and Adjacent Domains—Current Challenges and Research Gaps". In: *Journal of Software Engineering and Applications* 08 (Jan. 2015), pp. 499–519. DOI: 10.4236/jsea.2015.89048.

[Sal+23] Mikael Ebrahimi Salari, Eduard Paul Enoiu, Cristina Seceleanu, Wasif Afzal, and Filip Sebek. "Automating Test Generation of Industrial Control Software Through a PLC-to-Python Translation Framework and Pynguin". In: *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. 2023, pp. 431–440. DOI: 10.1109/APSEC60848.2023.00054.

[SB16] Klaus Schneider and Jens Brandt. "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems". In: Dordrecht: Springer Science+Business Media, Jan. 2016, pp. 1–30. DOI: 10.1007/978-94-017-7358-4_3-1.

[Sch+18] Alexander Schulz-Rosengarten, Reinhard Von Hanxleden, Frédéric Mallet, Robert De Simone, and Julien Deantoni. "Time in SCCharts". In: *2018 Forum on Specification Design Languages (FDL)*. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524111.

[Sch09] Klaus Schneider. *The Synchronous Programming Language Quartz*. Internal Report 375. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, Dec. 2009.

[Sch14] Bernd Schröder. *Steuerungstechnik für Ingenieure*. de. 2014th ed. essentials. Wiesbaden, Germany: Springer Fachmedien, Sept. 2014.

[Sch19] Karl Schmitt. *SPS-Programmierung mit ST: nach IEC 61131 mit CoDeSys und mit Hinweisen zu STEP 7 im TIA-Portal*. de. Nov. 2019.

[Shi+24] Jianqi Shi, Yinghao Chen, Qin Li, Yanhong Huang, Yang Yang, and Mengyan Zhao. "Automated Test Cases Generator for IEC 61131-3 Structured Text Based Dynamic Symbolic Execution". In: *IEEE Transactions on Computers* 73 (2024), pp. 1048–1059. URL: https://api.semanticscholar.org/CorpusID:266895399.

[SLH16] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. "Model Extraction of Legacy C Code in SCCharts". In: *Electronic Communications of the EASST* 74 (Oct. 2016). DOI: 10.14279/tuj.eceasst.74.

[SS06] Klaus Schneider and Tobias Schuele. "A Framework for Verifying and Implementing Embedded Systems." In: *MBMV*. 2006, pp. 242–247.

[STF12] Doaa Soliman, Kleanthis Thramboulidis, and Georg Frey. "Function Block Diagram to UPPAAL Timed Automata Transformation Based on Formal Models". In: *IFAC Proceedings Volumes* 45 (2012), pp. 1653–1659.

[Sun+08] Christoph Sunder, Monika Wenger, Christian Hanni, Ivo Gosetti, Heinrich Steininger, and Josef Fritsche. "Transformation of existing IEC 61131-3 automation projects into control logic according to IEC 61499". In: *2008 IEEE International Conference on Emerging Technologies and Factory Automation*. IEEE, 2008, pp. 369–376. ISBN: 1424415063. DOI: 10.1109/ETFA.2008.4638420.

[Tap15] Herbert Tapken. *LOGO! - Lösungen*. de. 5th ed. Haan, Germany: Europa-Lehrmittel, Jan. 2015.

[TF11] Kleanthis Thramboulidis and Georg Frey. "Towards a Model-Driven IEC 61131-Based Development Process in Industrial Automation". In: *Journal of Software Engineering and Applications* 04 (04 2011), pp. 217–226. ISSN: 1945-3116. DOI: 10.4236/jsea.2011.44024M4-Citavi.

[Thr13] Kleanthis Thramboulidis. "IEC 61499 vs. 61131: A Comparison Based on Misperceptions". In: *Journal of Software Engineering and Applications* 06.08 (2013), pp. 405–415. ISSN: 1945-3124. DOI: 10.4236/jsea.2013.68050. URL: http://dx.doi.org/10.4236/jsea.2013.68050.

[VK02] Norbert Völker and Bernd Johann Krämer. "Automated Verification of Function Block Based Industrial Control Systems". In: *Sci. Comput. Program.* 42 (Jan. 2002), pp. 101–113. DOI: 10.1016/S1571-0661(04)00135-5.

[Wen+09a] Monika Wenger, Alois Zoitl, Christoph Sünder, and Heinrich Steininger. "Semantic Correct Transformation of IEC 61131-3 Models into the IEC 61499 Standard". In: *2009 IEEE Conference on Emerging Technologies Factory Automation*. IEEE, Sept. 2009, pp. 1–7. ISBN: 978-1-4244-2727-7. DOI: 10.1109/ETFA.2009.5347144.

[Wen+09b] Monika Wenger, Alois Zoitl, Christoph Sünder, and Heinrich Steininger. "Transformation of IEC 61131-3 to IEC 61499 based on a model driven development approach". In: *IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2009, pp. 715–720. ISBN: 9781424437603. DOI: 10.1109/INDIN.2009.5195891.

[WFV09] Awang Noor Indra Wardana, Jens Folmer, and Birgit Vogel-Heuser. "Automatic program verification of continuous function chart based on model checking". In: *IECON Proceedings (Industrial Electronics Conference)*. IEEE, Dec. 2009, pp. 2422–2427. DOI: 10.1109/IECON.2009.5415231.

[Wir71] Niklaus Wirth. "The Programming Language Pascal". In: *Acta Inf.* 1.1 (Mar. 1971), pp. 35–63. ISSN: 0001-5903. DOI: 10.1007/BF00264291. URL: https://doi.org/10.1007/BF00264291.

[WS20] Marcel Christian Werner and Klaus Schneider. "Reengineering Programmable Logic Controllers Using Synchronous Programming Languages". In: *Forum on Specification and Design Languages (FDL)*. Work in Progress. Kiel, Germany, 2020.

[WS21] Marcel Christian Werner and Klaus Schneider. "Translation of Continuous Function Charts to Imperative Synchronous Quartz Programs". In: *2021 19th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2021, pp. 104–110. DOI: 10.1145/3487212.3487338.

[WS22] Marcel Christian Werner and Klaus Schneider. “From IEC 61131-3 Function Block Diagrams to Sequentially Constructive Statecharts”. In: *2022 Forum on Specification & Design Languages (FDL)*. 2022, pp. 1–8. DOI: 10.1109/FDL56239.2022.9925656.

[WS23] Marcel Christian Werner and Klaus Schneider. “Formal Methods-Based Optimization of Dataflow Models with Translation to Synchronous Models”. In: *2023 Forum on Specification & Design Languages (FDL)*. 2023, pp. 1–8. DOI: 10.1109/FDL59689.2023.10272138.

[WS24a] Marcel Christian Werner and Klaus Schneider. “From Imperative Sequential Structured Text Models to Synchronous Quartz and Sequentially Constructive Models”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. ITG-Fachbericht. Kaiserslautern, Germany: VDE, 2024, pp. 164–174.

[WS24b] Marcel Christian Werner and Klaus Schneider. “PLCreX – Open-Source Project for Simplification, Transformation, Analysis, and Validation of Programmable Logic Controllers”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. ITG-Fachbericht. Kaiserslautern, Germany: VDE, 2024, pp. 182–185.

[WV04] Daniel Witsch and Birgit Vogel-Heuser. “Automatische Codegenerierung aus der UML für die IEC 61131-3”. In: *Eingebettete Systeme*. Ed. by Peter Holleczek and Birgit Vogel-Heuser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 9–18. ISBN: 978-3-642-18594-6.

[WV09] Daniel Witsch and Birgit Vogel-Heuser. “Close integration between UML and IEC 61131-3: New possibilities through object-oriented extensions”. In: *2009 IEEE Conference on Emerging Technologies & Factory Automation* (2009), pp. 1–6. URL: https://api.semanticscholar.org/CorpusID:15055648.

[WZ07] Günter Wellenreuther and Dieter Zastrow. *Automatisieren mit SPS Übersichten und Übungsaufgaben - Von Grundverknüpfungen bis Ablaufsteuerungen: STEP 7-Programmierung, Lösungsmethoden, Lernaufgaben, Kontrollaufgaben, Lösungen, Beispiele zur Anlagensimulation*. Berlin Heidelberg New York: Springer-Verlag, 2007. ISBN: 978-3-834-89210-2.

[WZ12] Monika Wenger and Alois Zoitl. “Re-use of IEC 61131-3 structured text for IEC 61499”. In: Mar. 2012. DOI: 10.1109/ICIT.2012.6209917.

[Xio+20] Jiawen Xiong, Gang Zhu, Yanhong Huang, and Jianqi Shi. "A user-friendly verification approach for IEC 61131-3 PLC programs". In: *Electronics (Switzerland)* 9.4 (Apr. 2020). ISSN: 20799292. DOI: 10.3390/electronics9040572.

[YKL13] Junbeom Yoo, Eui Sub Kim, and Jang Soo Lee. "A behavior-preserving translation from FBD design to C implementation for reactor protection system software". In: *Nuclear Engineering and Technology* 45.4 (2013), pp. 489–504. ISSN: 1738-5733. DOI: 10.5516/NET.04.2012.085.

[Yoo+07] Li Hsien Yoong, Partha Roop, Valeriy Vyatkin, and Zoran Salcic. "Synchronous Execution of IEC 61499 Function Blocks Using Esterel". In: *2007 5th IEEE International Conference on Industrial Informatics*. Vol. 2. 2007, pp. 1189–1194. DOI: 10.1109/INDIN.2007.4384944.

[ZSM11] Justyna Zander, Ina Schieferdecker, and Pieter Mosterman. *Model-Based Testing for Embedded Systems*. Sept. 2011. ISBN: 9781439818459.

Appendix A

Detailed Syntax and Semantics

Contents

A.1. IEC 61131-3 FBDs and ST Models

A.1.1. POU Variants and Declaration

Syntax of POU elements

- $\delta_\omega(\omega^{prg}_{pou}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{PROGRAM}\ a_n(\omega^{prg}_{pou}) \\ \quad [\Delta_{idcl}(\omega^{prg}_{pou})]\quad [\Delta_{vdcl}(\omega^{prg}_{pou})]\quad [\Sigma(\omega^{prg}_{pou})] \\ \texttt{END\_PROGRAM} \end{array} \right\}$
- $\delta_\omega(\omega^{fb}_{pou}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{FUNCTION\_BLOCK}\ a_n(\omega^{fb}_{pou}) \\ \quad [\Delta_{idcl}(\omega^{fb}_{pou})]\quad [\Delta_{vdcl}(\omega^{fb}_{pou})]\quad [\Sigma(\omega^{fb}_{pou})] \\ \texttt{END\_FUNCTION\_BLOCK} \end{array} \right\}$
- $\delta_\omega(\omega^{fun}_{pou}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{FUNCTION}\ a_n(\omega^{fun}_{pou})\ [\texttt{:}\ \alpha] \\ \quad [\Delta_{idcl}(\omega^{fun}_{pou})]\quad [\Delta_{vdcl}(\omega^{fun}_{pou})]\quad [\Sigma(\omega^{fun}_{pou})] \\ \texttt{END\_FUNCTION} \end{array} \right\}$

Semantics of POU elements

- $[\![\delta_\omega(\omega_{pou}^{prg})]\!]_\xi \stackrel{\text{def}}{=}$ { defines a POU element ω_{pou}^{prg} with $\Sigma(\omega_{pou}^{prg})$, started at time t with initial conditions set by $\Delta_{idcl}(\omega_{pou}^{prg})$ and $\Delta_{vdcl}(\omega_{pou}^{prg})$ when invoked, terminating at time $t+\theta$ and preserving internal state across invocations by a resource, i.e., ω_{pou}^{prg} has memory }

- $[\![\delta_\omega(\omega_{pou}^{fb})]\!]_\xi \stackrel{\text{def}}{=}$ { defines a POU element ω_{pou}^{fb} with $\Sigma(\omega_{pou}^{fb})$, started at time t with initial conditions set by $\Delta_{idcl}(\omega_{pou}^{fb})$ and $\Delta_{vdcl}(\omega_{pou}^{fb})$ when invoked, terminating at time $t+\theta$ and preserving internal state across invocations by another POU element, i.e., ω_{pou}^{fb} has memory }

- $[\![\delta_\omega(\omega_{pou}^{fun})]\!]_\xi \stackrel{\text{def}}{=}$ { defines an ST model element ω_{pou}^{fun} with $\Sigma(\omega_{pou}^{fun})$ and optional output of data type α, started at time t with initial conditions set by $\Delta_{idcl}(\omega_{pou}^{fun})$ and $\Delta_{vdcl}(\omega_{pou}^{fun})$ when invoked, terminating at time $t+\theta$ and not preserving internal state after invocation by another POU element, i.e., ω_{pou}^{fun} has no memory }

A.1.2. POU Interfaces

Syntax of POU interfaces

- $\Delta_{in}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{VAR\_INPUT} \\ \quad x_1 \ : \ \alpha_1^{[+]}\texttt{[:=}w_1\texttt{];} \\ \quad \vdots \\ \quad x_n \ : \ \alpha_n^{[+]}\texttt{[:=}w_n\texttt{];} \\ \texttt{END\_VAR} \end{array} \right\}$

- $\Delta_{out}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{VAR\_OUTPUT} \\ \quad x_1 \ : \ \alpha_1^{[+]}\texttt{[:=}w_1\texttt{];} \\ \quad \vdots \\ \quad x_n \ : \ \alpha_n^{[+]}\texttt{[:=}w_n\texttt{];} \\ \texttt{END\_VAR} \end{array} \right\}$

- $\Delta_{inout}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{VAR\_IN\_OUT} \\ \quad x_1 \ : \ \alpha_1^{[+]}\texttt{;} \\ \quad \vdots \\ \quad x_n \ : \ \alpha_n^{[+]}\texttt{;} \\ \texttt{END\_VAR} \end{array} \right\}$

Semantics of POU interfaces

- $[\![\Delta_{in}(\omega_{pou}^{\varphi})]\!]_\xi \stackrel{\text{def}}{=}$ { defines a set of variables $x_1, \ldots, x_n$ with corresponding data types $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. These variables are assigned to optional predefined default values $[\![w_1]\!]_\xi, \ldots, [\![w_n]\!]_\xi$ or to externally supplied values at time t when ω_{pou}^{φ} is invoked. Their values cannot be modified until ω_{pou}^{φ} terminates at time $t+\theta$, i.e., all instructions of ω_{pou}^{φ} have been processed in the current PLC cycle. }

- $[\![\Delta_{out}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=}$ { defines a set of variables $x_1, \ldots, x_n$ with corresponding data types $[\![\alpha_1^{[+]}]\!]_{\xi}, \ldots, [\![\alpha_n^{[+]}]\!]_{\xi}$. These variables are assigned to optional predefined default values $[\![w_1]\!]_{\xi}, \ldots, [\![w_n]\!]_{\xi}$ at time t when ω^{φ}_{pou} is invoked, and passed to invoking element with final computed values when ω^{φ}_{pou} terminates at time $t+\theta$, i.e., all instructions of ω^{φ}_{pou} have been processed in the current PLC cycle. }

- $[\![\Delta_{inout}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=}$ { defines a set of variables $x_1, \ldots, x_n$ with corresponding data types $[\![\alpha_1^{[+]}]\!]_{\xi}, \ldots, [\![\alpha_n^{[+]}]\!]_{\xi}$. These variables are assigned to externally supplied values at time t when ω^{φ}_{pou} is invoked. Their values can be modified until ω^{φ}_{pou} terminates at time $t+\theta$, i.e., all instructions of ω^{φ}_{pou} have been processed in the current PLC cycle, and then passed to invoking element }

A.1.3. Local Variables in POUs

Syntax of local variables in POUs

- $\Delta_{local}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{VAR} \\ \quad x_1 \ : \ \alpha_1^{[+]}\texttt{[:=}w_1\texttt{];} \\ \quad \vdots \\ \quad x_n \ : \ \alpha_n^{[+]}\texttt{[:=}w_n\texttt{];} \\ \texttt{END\_VAR} \end{array} \right\}$

- $\Delta_{inst}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{VAR} \\ \quad x_1 \ : \ a_n(k_1)\texttt{;} \\ \quad \vdots \\ \quad x_n \ : \ a_n(k_n)\texttt{;} \\ \texttt{END\_VAR} \end{array} \right\}$

Semantics of local variables in POUs

- $[\![\Delta_{local}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=}$ { defines a set of variables $x_1, \ldots, x_n$ with corresponding data types $[\![\alpha_1^{[+]}]\!]_{\xi}, \ldots, [\![\alpha_n^{[+]}]\!]_{\xi}$. Depending on the POU variant, the variables keep their values from previous invocation or are assigned to optional predefined default values $[\![w_1]\!]_{\xi}, \ldots, [\![w_n]\!]_{\xi}$ at time t when ω^{φ}_{pou} is invoked. These variables can be modified and processed locally until ω^{φ}_{pou} terminates at time $t+\theta$, i.e., all instructions of ω^{φ}_{pou} have been processed in the current PLC cycle. }

- $[\![\Delta_{inst}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=}$ { defines a set of instances $x_1, \ldots, x_n$ of corresponding POUs with memory $[\![a_n(k_1)]\!]_{\xi}, \ldots, [\![a_n(k_n)]\!]_{\xi}$, that can be processed locally between ω^{φ}_{pou} was invoked at time t and ω^{φ}_{pou} terminates at time $t+\theta$, i.e., all instructions of ω^{φ}_{pou} have been processed in the current PLC cycle. }

A.1.4. Elementary IEC 61131-3 Data Types and Fields

Syntax of elementary IEC 61131-3 data types and fields

Syntax of bit vector data types $\alpha_{bv}(\omega^{\varphi}_{pou}) \in \mathcal{A}_{bv}(\omega^{\varphi}_{pou})$:

- $\alpha^{bool}_{bv}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{BOOL}\}$ denotes Boolean values
- $\alpha^{byte}_{bv}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{BYTE}\}$ denotes single byte bit masks
- $\alpha^{word}_{bv}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{WORD}\}$ denotes two byte bit masks

Syntax of integer data types $\alpha_i(\omega^{\varphi}_{pou}) \in \mathcal{A}_i(\omega^{\varphi}_{pou})$:

- $\alpha^{int}_{i}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{INT}\}$ denotes bounded signed integers
- $\alpha^{dint}_{i}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{DINT}\}$ denotes bounded signed double integers
- $\alpha^{uint}_{i}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{UINT}\}$ denotes bounded unsigned integers
- $\alpha^{udint}_{i}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{UDINT}\}$ denotes bounded unsigned double integers

Syntax of floating-point data types $\alpha_r(\omega^{\varphi}_{pou}) \in \mathcal{A}_r(\omega^{\varphi}_{pou})$:

- $\alpha^{real}_{r}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{REAL}\}$ denotes floating-point values

Syntax of numeric data types $\alpha_{num}(\omega^{\varphi}_{pou}) \in \mathcal{A}_i(\omega^{\varphi}_{pou}) \bigcup \mathcal{A}_r(\omega^{\varphi}_{pou})$:

- $\alpha_{num}(\omega^{\varphi}_{pou}) \in \{\alpha_i(\omega^{\varphi}_{pou}), \alpha_r(\omega^{\varphi}_{pou})\}$

Syntax of duration data types $\alpha_{dur}(\omega^{\varphi}_{pou}) \in \mathcal{A}_{dur}(\omega^{\varphi}_{pou})$:

- $\alpha^{time}_{dur}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{TIME}\}$ denotes interval value in milliseconds

Syntax of data type fields $\alpha^{+}(\omega^{\varphi}_{pou}) \in \mathcal{A}^{+}(\omega^{\varphi}_{pou})$:

- $\alpha^{+}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{ARRAY[0..}n\texttt{] OF } \alpha(\omega^{\varphi}_{pou})\}$ denotes arrays

Semantics of elementary IEC 61131-3 data types and fields

Semantics of bit vector data types $\alpha_{bv}(\omega^{\varphi}_{pou}) \in \mathcal{A}_{bv}(\omega^{\varphi}_{pou})$:

- $[\![\alpha^{bool}_{bv}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{false, true\}, false\rangle$
- $[\![\alpha^{byte}_{bv}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{\texttt{00}_{hex}, \texttt{ff}_{hex}\}, \texttt{00}_{hex}\rangle$
- $[\![\alpha^{word}_{bv}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{\texttt{0000}_{hex}, \texttt{ffff}_{hex}\}, \texttt{0000}_{hex}\rangle$

Semantics of integer data types $\alpha_i(\omega^{\varphi}_{pou}) \in \mathcal{A}_i(\omega^{\varphi}_{pou})$:

- $[\![\alpha^{int}_{i}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{-2^{15}, 2^{15}-1\}, 0\rangle$
- $[\![\alpha^{dint}_{i}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{-2^{31}, 2^{31}-1\}, 0\rangle$
- $[\![\alpha^{uint}_{i}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{0, 2^{16}-1\}, 0\rangle$

- $[\![\alpha_i^{udint}(\omega_{pou}^{\varphi})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{0, 2^{32}-1\}, 0\rangle$

Semantics of floating-point data types $\alpha_r(\omega_{pou}^{\varphi}) \in \mathcal{A}_r(\omega_{pou}^{\varphi})$:

- $[\![\alpha_r^{real}(\omega_{pou}^{\varphi})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{\text{single precision floating-point (32 Bits)}\}, 0\rangle$

Semantics of duration data types $\alpha_{dur}(\omega_{pou}^{\varphi}) \in \mathcal{A}_{dur}(\omega_{pou}^{\varphi})$:

- $[\![\alpha_{dur}^{time}(\omega_{pou}^{\varphi})]\!]_{\xi} \stackrel{\text{def}}{=} \langle\{0, 2^{32}-1\}, 0\rangle$

Semantics of data type fields $\alpha^{+}(\omega_{pou}^{\varphi}) \in \mathcal{A}^{+}(\omega_{pou}^{\varphi})$:

- $[\![\alpha^{+}(\omega_{pou}^{\varphi})]\!]_{\xi} \stackrel{\text{def}}{=} \{$ group with $n+1$ elements of data type $\alpha(\omega_{pou}^{\varphi})\}$

A.1.5. Expressions in POUs

Syntax of expressions in POUs

Syntax of constants and general expressions $\tau_{misc}(\omega_{pou}^{\varphi}) \in \mathcal{T}_{misc}(\omega_{pou}^{\varphi})$:

- $\tau_{misc}^{cst}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \{\texttt{x}\}$ denotes a value constant x
- $\tau_{misc}^{id}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \{\texttt{y}\}$ denotes an identifier y
- $\tau_{misc}^{br}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \{(\tau(\omega_{pou}^{\varphi}))\}$ denotes a bracket
- $\tau_{misc}^{true}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \{\texttt{TRUE}\}$ denotes a `true`-constant
- $\tau_{misc}^{false}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \{\texttt{FALSE}\}$ denotes a `false`-constant
- $\tau_{misc}^{arr}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \{x\texttt{[}n\texttt{]}\}$ denotes access to index n of array x
- $\tau_{misc}^{inv}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \{x\texttt{.}y\}$ denotes access to port y of instance x

Syntax of comparison operators $\tau_{comp}(\omega_{pou}^{\varphi}) \in \mathcal{T}_{comp}(\omega_{pou}^{\varphi})$:

- $\tau_{comp}^{eq}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \begin{cases}\{\pi_1 \texttt{ = } \pi_2\}, & \text{if } \varphi = st\\ \{\texttt{EQ(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\}\end{cases}$ denotes equality
- $\tau_{comp}^{ne}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \begin{cases}\{\pi_1 \texttt{ <> } \pi_2\}, & \text{if } \varphi = st\\ \{\texttt{NE(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\}\end{cases}$ denotes inequality
- $\tau_{comp}^{gt}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \begin{cases}\{\pi_1 \texttt{ > } \pi_2\}, & \text{if } \varphi = st\\ \{\texttt{GT(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\}\end{cases}$ denotes greater than
- $\tau_{comp}^{ge}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \begin{cases}\{\pi_1 \texttt{ >= } \pi_2\}, & \text{if } \varphi = st\\ \{\texttt{GE(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\}\end{cases}$ denotes greater than/equal to
- $\tau_{comp}^{lt}(\omega_{pou}^{\varphi}) \stackrel{\text{def}}{=} \begin{cases}\{\pi_1 \texttt{ < } \pi_2\}, & \text{if } \varphi = st\\ \{\texttt{LT(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\}\end{cases}$ denotes lower than

- $\tau^{le}_{comp}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\pi_1 \texttt{ <= } \pi_2\}, & \text{if } \varphi = st \\ \{\texttt{LE(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes lower than/equal to

Syntax of arithmetic operators $\tau_{arith}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{arith}(\omega^{\varphi}_{pou})$:

- $\tau^{mul}_{arith}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\eta_1 \texttt{ * } \eta_2\}, & \text{if } \varphi = st \\ \{\texttt{MUL(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes a multiplication
- $\tau^{div}_{arith}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\eta_1 \texttt{ / } \eta_2\}, & \text{if } \varphi = st \\ \{\texttt{DIV(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes a division
- $\tau^{add}_{arith}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\eta_1 \texttt{ + } \eta_2\}, & \text{if } \varphi = st \\ \{\texttt{ADD(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes an addition
- $\tau^{sub}_{arith}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\eta_1 \texttt{ - } \eta_2\}, & \text{if } \varphi = st \\ \{\texttt{SUB(}\pi_1\texttt{,}\pi_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes a subtraction
- $\tau^{expt}_{arith}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\eta_{r,1} \texttt{ ** } \eta_2\}, & \text{if } \varphi = st \\ \{\texttt{EXPT(}\eta_{r,1}\texttt{,}\eta_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes an exponential function
- $\tau^{mod}_{arith}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\eta^i_1 \texttt{ MOD } \eta^i_2\}, & \text{if } \varphi = st \\ \{\texttt{MOD(}\eta^i_1\texttt{,}\eta^i_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes a modulo function
- $\tau^{um}_{arith}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{-}\eta_1\}$ denotes an unary minus

Syntax of bitwise operators $\tau_{bv}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{bv}(\omega^{\varphi}_{pou})$:

- $\tau^{and}_{bv}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\lambda_1 \texttt{ \& } \ldots \texttt{ \& } \lambda_n\}, & \text{if } \varphi = st \\ \{\lambda_1 \texttt{ AND } \ldots \texttt{ AND } \lambda_n\}, & \text{if } \varphi = st \\ \{\texttt{AND(}\lambda_1\texttt{,}\ldots\texttt{,}\lambda_n\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes a conjunction
- $\tau^{or}_{bv}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\lambda_1 \texttt{ OR } \ldots \texttt{ OR } \lambda_n\}, & \text{if } \varphi = st \\ \{\texttt{OR(}\lambda_1\texttt{,}\ldots\texttt{,}\lambda_n\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes a disjunction
- $\tau^{xor}_{bv}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\lambda_1 \texttt{ XOR } \lambda_2\}, & \text{if } \varphi = st \\ \{\texttt{XOR(}\lambda_1\texttt{,}\lambda_2\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes an exclusive or
- $\tau^{not}_{bv}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \begin{cases} \{\texttt{NOT } \lambda_1\}, & \text{if } \varphi = st \\ \{\texttt{NOT(}\lambda_1\texttt{)}\}, & \text{if } \varphi \in \{st, fbd\} \end{cases}$ denotes a negation

Syntax of conditional operators $\tau_{cond}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{cond}(\omega^{\varphi}_{pou})$:

- $\tau^{sel}_{cond}(\omega^{\varphi}_{pou}) \stackrel{\text{def}}{=} \{\texttt{SEL(}\lambda^b\texttt{,}\pi_1\texttt{,}\pi_2\texttt{)}\}$ denotes a conditional operator

Type system of expressions in POUs

Type system of comparison operators $\tau^{\gamma}_{comp}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{comp}(\omega^{\varphi}_{pou})$, considering $\gamma \in \{eq, ne, gt, ge, lt, le\}$:

- $$\frac{\pi_1 : \alpha(\omega^{\varphi}_{pou}) \quad \pi_2 : \alpha(\omega^{\varphi}_{pou})}{\tau^{\gamma}_{comp}(\omega^{\varphi}_{pou}) : \alpha^{bool}_{bv}(\omega^{\varphi}_{pou})}$$

Type system of arithmetic operators $\tau^{\gamma}_{arith}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{arith}(\omega^{\varphi}_{pou})$, considering $\gamma \in \{mul, div, add, sub, um\}$:

- $$\frac{\eta_1 : \alpha_{num}(\omega^{\varphi}_{pou}) \quad [\eta_2 : \alpha_{num}(\omega^{\varphi}_{pou})]}{\tau^{\gamma}_{arith}(\omega^{\varphi}_{pou}) : \alpha_{num}(\omega^{\varphi}_{pou})}$$

Type system of modulo operator $\tau^{mod}_{arith}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{arith}(\omega^{\varphi}_{pou})$:

- $$\frac{\eta^i_1 : \alpha_i(\omega^{\varphi}_{pou}) \quad \eta^i_2 : \alpha_i(\omega^{\varphi}_{pou})}{\tau^{mod}_{arith}(\omega^{\varphi}_{pou}) : \alpha_i(\omega^{\varphi}_{pou})}$$

Type system of exponential function operator $\tau^{expt}_{arith}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{arith}(\omega^{\varphi}_{pou})$:

- $$\frac{\eta^r_1 : \alpha_r(\omega^{\varphi}_{pou}) \quad \eta_2 : \alpha_{num}(\omega^{\varphi}_{pou})}{\tau^{expt}_{arith}(\omega^{\varphi}_{pou}) : \alpha_r(\omega^{\varphi}_{pou})}$$

Type system of bitwise operators $\tau^{\gamma}_{bv}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{bv}(\omega^{\varphi}_{pou})$, considering $\gamma \in \{and, or, xor, not\}$:

- $$\frac{\lambda_1 : \alpha_{bv}(\omega^{\varphi}_{pou}) \quad [\lambda_2 : \alpha_{bv}(\omega^{\varphi}_{pou}) \quad \cdots \quad \lambda_n : \alpha_{bv}(\omega^{\varphi}_{pou})]}{\tau^{\gamma}_{bv}(\omega^{\varphi}_{pou}) : \alpha^{bool}_{bv}(\omega^{\varphi}_{pou})}$$

Type system of conditional operator $\tau^{sel}_{cond}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{cond}(\omega^{\varphi}_{pou})$:

- $$\frac{\lambda^b : \alpha^{bool}_{bv}(\omega^{\varphi}_{pou}) \quad \pi_1 : \alpha(\omega^{\varphi}_{pou}) \quad \pi_2 : \alpha(\omega^{\varphi}_{pou})}{\tau^{sel}_{cond}(\omega^{\varphi}_{pou}) : \alpha(\omega^{\varphi}_{pou})}$$

Semantics of expressions in POUs

Semantics of constants and general expressions $\tau_{misc}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{misc}(\omega^{\varphi}_{pou})$:

- $[\![\tau^{cst}_{misc}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![x]\!]_{\xi}$
- $[\![\tau^{id}_{misc}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} y$
- $[\![\tau^{br}_{misc}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} (\tau_{misc}(\omega^{\varphi}_{pou}))$
- $[\![\tau^{true}_{misc}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} true$
- $[\![\tau^{false}_{misc}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} false$
- $[\![\tau^{arr}_{misc}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![x\,[n]]\!]_{\xi}$
- $[\![\tau^{inv}_{misc}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![k\,.\,x]\!]_{\xi}$

Semantics of comparison operators $\tau_{comp}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{comp}(\omega^{\varphi}_{pou})$:

- $[\![\tau^{eq}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\pi_1]\!]_{\xi} = [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{ne}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\pi_1]\!]_{\xi} \neq [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{gt}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\pi_1]\!]_{\xi} > [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{ge}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\pi_1]\!]_{\xi} >= [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{lt}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\pi_1]\!]_{\xi} < [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{le}_{comp}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\pi_1]\!]_{\xi} <= [\![\pi_2]\!]_{\xi}$

Semantics of arithmetic operators $\tau_{arith}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{arith}(\omega^{\varphi}_{pou})$:

- $[\![\tau^{mul}_{arith}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\eta_1]\!]_{\xi} \cdot [\![\eta_2]\!]_{\xi}$
- $[\![\tau^{div}_{arith}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \frac{[\![\eta_1]\!]_{\xi}}{[\![\eta_2]\!]_{\xi}}$
- $[\![\tau^{add}_{arith}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\eta_1]\!]_{\xi} + [\![\eta_2]\!]_{\xi}$
- $[\![\tau^{sub}_{arith}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\eta_1]\!]_{\xi} - [\![\eta_2]\!]_{\xi}$
- $[\![\tau^{expt}_{arith}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\eta^r_1]\!]^{[\![\eta_2]\!]_{\xi}}_{\xi}$
- $[\![\tau^{mod}_{arith}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\eta^i_1]\!]_{\xi} \; mod \; [\![\eta^i_2]\!]_{\xi}$
- $[\![\tau^{um}_{arith}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} -[\![\eta_1]\!]_{\xi}$

Semantics of Boolean operators $\tau_{bv}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{bv}(\omega^{\varphi}_{pou})$:

- $[\![\tau^{and}_{bv}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\lambda_1]\!]_{\xi} \wedge \cdots \wedge [\![\lambda_n]\!]_{\xi}$
- $[\![\tau^{or}_{bv}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\lambda_1]\!]_{\xi} \vee \cdots \vee [\![\lambda_n]\!]_{\xi}$
- $[\![\tau^{xor}_{bv}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} [\![\lambda_1]\!]_{\xi} \oplus [\![\lambda_2]\!]_{\xi}$
- $[\![\tau^{not}_{bv}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \neg[\![\lambda_1]\!]_{\xi}$

Semantics of conditional operators $\tau^{sel}_{cond}(\omega^{\varphi}_{pou}) \in \mathcal{T}_{cond}(\omega^{\varphi}_{pou})$:

- $[\![\tau^{sel}_{cond}(\omega^{\varphi}_{pou})]\!]_{\xi} \stackrel{\text{def}}{=} \begin{cases} [\![\pi_1]\!]_{\xi}, & \text{if } [\![\lambda^b]\!]_{\xi} = true \\ [\![\pi_2]\!]_{\xi}, & \text{otherwise} \end{cases}$

SOS transition rules of expressions in POUs

- $\langle \xi, \tau(\omega^{\varphi}_{pou}) \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{[\![\tau(\omega^{\varphi}_{pou})]\!]_{\xi}\}, true \rangle$

A.1.6. Conditions in ST Models

Syntax of conditions in ST models

- $\sigma^{it}_{cond}(\omega^{\varphi}_{st}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{IF } \lambda^b(\sigma^{it}_{cond}(\omega^{\varphi}_{st})) \texttt{ THEN } \Sigma_1(\sigma^{it}_{cond}(\omega^{\varphi}_{st})) \\ \texttt{END\_IF} \end{array} \right\}$

- $\sigma^{ite}_{cond}(\omega^{\varphi}_{st}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{IF } \lambda^b(\sigma^{ite}_{cond}(\omega^{\varphi}_{st})) \texttt{ THEN } \Sigma_1(\sigma^{ite}_{cond}(\omega^{\varphi}_{st})) \\ \texttt{ELSE } \Sigma_2(\sigma^{ite}_{cond}(\omega^{\varphi}_{st})) \texttt{ END\_IF} \end{array} \right\}$

SOS transition rules of conditions in ST models

SOS transition rules of $\sigma^{it}_{cond}(\omega^{\varphi}_{st}) \in \Sigma^{it}_{cond}(\omega^{\varphi}_{st})$:

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{IF } \lambda^b \texttt{ THEN } \Sigma_1 \\ \texttt{END\_IF} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{IF } \lambda^b \texttt{ THEN } \Sigma_1 \\ \texttt{END\_IF} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}$$

SOS transition rules of $\sigma^{ite}_{cond}(\omega^{\varphi}_{st}) \in \Sigma^{ite}_{cond}(\omega^{\varphi}_{st})$:

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{IF } \lambda^b \texttt{ THEN } \Sigma_1 \\ \texttt{ELSE } \Sigma_2 \texttt{ END\_IF} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_2, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{IF } \lambda^b \texttt{ THEN } \Sigma_1 \\ \texttt{ELSE } \Sigma_2 \texttt{ END\_IF} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_2, true \rangle}$$

A.1.7. Loops in ST Models

Syntax of loops in ST models

- $\sigma^{foot}_{loop}(\omega^{\varphi}_{st}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{REPEAT } \Sigma(\sigma^{foot}_{loop}(\omega^{\varphi}_{st})) \\ \texttt{UNTIL } \lambda^b(\sigma^{foot}_{loop}(\omega^{\varphi}_{st})) \end{array} \right\}$

- $\sigma^{head}_{loop}(\omega^{\varphi}_{st}) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b(\sigma^{head}_{loop}(\omega^{\varphi}_{st})) \texttt{ DO} \\ \Sigma(\sigma^{head}_{loop}(\omega^{\varphi}_{st})) \texttt{ END\_WHILE} \end{array} \right\}$

SOS transition rules of loops in ST models

SOS transition rules of $\sigma^{foot}_{loop}(\omega^{\varphi}_{st}) \in \Sigma^{foot}_{loop}(\omega^{\varphi}_{st})$:

- $$\frac{\langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle nothing, \mathcal{D}, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{REPEAT } \Sigma \\ \texttt{UNTIL } \lambda^b \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle nothing, \{ \mathcal{D}; \left\{ \begin{array}{l} \texttt{WHILE NOT } \lambda^b \texttt{ DO} \\ \mathcal{D} \texttt{ END\_WHILE} \end{array} \right\}, true \rangle}$$

SOS transition rules of $\sigma^{head}_{loop}(\omega^{\varphi}_{st}) \in \Sigma^{head}_{loop}(\omega^{\varphi}_{st})$:

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle nothing, \mathcal{D}, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b \texttt{ DO} \\ \Sigma \texttt{ END\_WHILE} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle nothing, \{ \mathcal{D}; \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b \texttt{ DO} \\ \mathcal{D} \texttt{ END\_WHILE} \end{array} \right\}, true \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false}{\langle \xi, \left\{ \begin{array}{l} \texttt{WHILE } \lambda^b \texttt{ DO} \\ \Sigma \texttt{ END\_WHILE} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle nothing, \{\}, true \rangle}$$

A.2. *Quartz* Models

A.2.1. *Quartz* Variants and Declaration

Definition A.1 (Syntax of *Quartz* elements). *A* Quartz *module is declared as follows [Sch09]:*

- $\delta_\omega(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} [\Delta_{imports}(\omega_{qrz})] \\ \texttt{module } a_n(\omega_{qrz})\texttt{([}\Delta_{idcl}(\omega_{qrz})\texttt{])\{} \\ \quad [\Delta_{vdcl}(\omega_{qrz})] \\ \quad [\Sigma(\omega_{qrz})] \\ \texttt{\}} \end{array} \right\}$

Definition A.2 (Semantics of *Quartz* elements). *The semantics of ω_{qrz} are defined as follows [Sch09]:*

- $[\![\delta_\omega(\omega_{qrz})]\!]_\xi \stackrel{def}{=} \{$ *defines a* Quartz *model element ω_{qrz} with $\Sigma(\omega_{qrz})$, started at time t with initial conditions set by $\Delta_{idcl}(\omega_{qrz})$ and $\Delta_{vdcl}(\omega_{qrz})$ when invoked, terminating at time $t + \theta$ and preserving internal state across* macro steps.$\}$

A.2.2. Module Imports

Definition A.3 (Syntax of *Quartz* model imports). *Imported* Quartz *models are grouped as a set $\Delta_{imports}(\omega_{qrz})$, whose syntax is defined as follows [Sch09]:*

- $\Delta_{imports}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{import } x_1\texttt{.*;} \\ \vdots \\ \texttt{import } x_n\texttt{.*;} \end{array} \right\}$

Definition A.4 (Semantics of *Quartz* model imports). *The semantics of* $\Delta_{imports}(\omega_{qrz})$ *are defined as follows [Sch09]:*

- $[\![\Delta_{imports}(\omega_{qrz})]\!]_\xi \stackrel{def}{=}$ { *defines a set of imported* Quartz *models* $x_1, \ldots, x_n$ *that can be instantiated and invoked by* ω_{qrz}. }

A.2.3. *Quartz* Interfaces

Definition A.5 (Syntax of *Quartz* interfaces). *The syntax of* $\Delta_{in}(\omega_{qrz})$, $\Delta_{out}(\omega_{qrz})$, *and* $\Delta_{inout}(\omega_{qrz})$ *is defined as follows [Sch09]:*

- $\Delta_{in}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{ll} \texttt{[event]} & \alpha_1^{[+]} \ \texttt{?}x_1 \\ \vdots & \\ \texttt{[event]} & \alpha_n^{[+]} \ \texttt{?}x_n \end{array} \right\}$
- $\Delta_{out}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{ll} \texttt{[event]} & \alpha_1^{[+]} \ \texttt{!}x_1 \\ \vdots & \\ \texttt{[event]} & \alpha_n^{[+]} \ \texttt{!}x_n \end{array} \right\}$
- $\Delta_{inout}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{ll} \texttt{[event]} & \alpha_1^{[+]} \ x_1 \\ \vdots & \\ \texttt{[event]} & \alpha_n^{[+]} \ x_n \end{array} \right\}$

Definition A.6 (Semantics of *Quartz* interfaces). *The semantics of* $\Delta_{in}(\omega_{qrz})$, $\Delta_{out}(\omega_{qrz})$, *and* $\Delta_{inout}(\omega_{qrz})$ *are defined as follows [Sch09]:*

- $[\![\Delta_{in}(\omega_{qrz})]\!]_\xi \stackrel{def}{=}$ { *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. *These variables are assigned to externally supplied values at time* t *when* ω_{qrz} *is invoked. Their values can be modified until* ω_{qrz} *terminates at time* $t + \theta$, *i.e., all instructions of* ω_{qrz} *have been processed. Variables classified as* event *variables are reset to their default values if they are not assigned in the current* macro step. }

- $[\![\Delta_{out}(\omega_{qrz})]\!]_\xi \stackrel{def}{=}$ { *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. *These variables can be updated until* ω_{qrz} *terminates at time* $t + \theta$, *and are passed to invoking element in each* macro step. *Variables classified as* event *variables are reset to their default values if they are not assigned in the current* macro step. }

- $[\![\Delta_{inout}(\omega_{qrz})]\!]_\xi \stackrel{def}{=}$ { *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. *These variables can be read and updated until* ω_{qrz} *terminates at time* $t + \theta$, *and are passed to invoking element in each* macro step. *Variables classified as* event *variables are reset to their default values if they are not assigned in the current* macro step. }

A.2.4. Local Variables in *Quartz* Models

Definition A.7 (Syntax of local variables in *Quartz* models). *The syntax of $\Delta_{local}(\omega_{qrz})$ is defined as follows:*

- $\Delta_{local}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[event]}\ \ \alpha_1^{[+]}\ \ x_1; \\ \vdots \\ \texttt{[event]}\ \ \alpha_n^{[+]}\ \ x_n; \end{array} \right\}$

Definition A.8 (Semantics of local variables in *Quartz* models). *The semantics of $\Delta_{local}(\omega_{qrz})$ are defined as follows:*

- $[\![\Delta_{local}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} \{$ *defines a set of variables $x_1, \ldots, x_n$ with corresponding data types $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. These variables can be read and updated internally in every* macro step *until ω_{qrz} terminates at time $t + \theta$. Variables classified as* event *variables are reset to their default values if they are not assigned in the current* macro step. $\}$

A.2.5. Elementary *Quartz* Data Types and Fields

Definition A.9 (Syntax of elementary *Quartz* data types and fields). *The syntax of $\alpha(\omega_{qrz}) \in \mathcal{A}(\omega_{qrz})$ and $\alpha^+(\omega_{qrz}) \in \mathcal{A}^+(\omega_{qrz})$ is defined as follows [Sch09]:*

Syntax of bit vector data types $\alpha_{bv}(\omega_{qrz}) \in \mathcal{A}_{bv}(\omega_{qrz})$:

- $\alpha_{bv}^{bool}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{bool}\}$ *denotes Boolean values*

Syntax of integer data types $\alpha_i(\omega_{qrz}) \in \mathcal{A}_i(\omega_{qrz})$:

- $\alpha_i^{int}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{int\{n\}}\}$ *denotes bounded signed integers*

Syntax of integer data types $\alpha_i(\omega_{qrz}) \in \mathcal{A}_i(\omega_{qrz})$:

- $\alpha_i^{uint}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{nat\{n\}}\}$ *denotes bounded unsigned integers*

Syntax of floating-point data types $\alpha_r(\omega_{qrz}) \in \mathcal{A}_r(\omega_{qrz})$:

- $\alpha_r^{real}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{real}\}$ *denotes floating-point values*

Syntax of numeric data types $\alpha_{num}(\omega_{qrz}) \in \mathcal{A}_i(\omega_{qrz}) \bigcup \mathcal{A}_r(\omega_{qrz})$:

- $\alpha_{num}(\omega_{qrz}) \in \{\alpha_i(\omega_{qrz}), \alpha_r(\omega_{qrz})\}$

Syntax of duration data types $\alpha_{dur}(\omega_{qrz}) \in \mathcal{A}_{dur}(\omega_{qrz})$:

- $\alpha_{dur}^{time}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{nat}\}$ *denotes interval value in unbounded integer*

Syntax of data type fields $\alpha^+(\omega_{qrz}) \in \mathcal{A}^+(\omega_{qrz})$:

- $\alpha^{+}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{[n]}\alpha(\omega_{qrz})\ \ y\}$ *denotes arrays*

Definition A.10 (Semantics of elementary *Quartz* data types and fields). *The semantics of* $\alpha(\omega_{qrz}) \in \mathcal{A}(\omega_{qrz})$ *and* $\alpha^{+}(\omega_{qrz}) \in \mathcal{A}^{+}(\omega_{qrz})$ *are defined as follows [Sch09]:*

Semantics of bit vector data types $\alpha_{bv}(\omega_{qrz}) \in \mathcal{A}_{bv}(\omega_{qrz})$*:*

- $[\![\alpha_{bv}^{bool}(\omega_{qrz})]\!]_{\xi} \stackrel{def}{=} \langle\{false, true\}, false\rangle$

Semantics of integer data types $\alpha_{i}(\omega_{qrz}) \in \mathcal{A}_{i}(\omega_{qrz})$*:*

- $[\![\alpha_{i}^{int}(\omega_{qrz})]\!]_{\xi} \stackrel{def}{=} \langle\{-n, n-1\}, 0\rangle$
- $[\![\alpha_{i}^{uint}(\omega_{qrz})]\!]_{\xi} \stackrel{def}{=} \langle\{0, n-1\}, 0\rangle$

Semantics of floating-point data types $\alpha_{r}(\omega_{qrz}) \in \mathcal{A}_{r}(\omega_{qrz})$*:*

- $[\![\alpha_{r}^{real}(\omega_{qrz})]\!]_{\xi} \stackrel{def}{=} \langle\{\textit{single precision floating-point (32 Bits)}\}, 0\rangle$

Semantics of duration data types $\alpha_{dur}(\omega_{qrz}) \in \mathcal{A}_{dur}(\omega_{pou})$*:*

- $[\![\alpha_{dur}^{time}(\omega_{qrz})]\!]_{\xi} \stackrel{def}{=} \langle\mathbb{N}, 0\rangle$

Semantics of data type fields $\alpha^{+}(\omega_{qrz}) \in \mathcal{A}^{+}(\omega_{qrz})$*:*

- $[\![\alpha^{+}(\omega_{qrz})]\!]_{\xi} \stackrel{def}{=} \{$ *group with* n *elements of data type* $\alpha(\omega_{qrz})\}$

A.2.6. Expressions in *Quartz* Models

Definition A.11 (Syntax of expressions in *Quartz* models). *The syntax of* $\mathcal{T}_{misc}(\omega_{qrz})$*,* $\mathcal{T}_{comp}(\omega_{qrz})$*,* $\mathcal{T}_{arith}(\omega_{qrz})$*,* $\mathcal{T}_{bv}(\omega_{qrz})$*, and* $\mathcal{T}_{cond}(\omega_{qrz})$ *is defined as follows [Sch09]:*

Syntax of constants and general expressions $\tau_{misc}(\omega_{qrz}) \in \mathcal{T}_{misc}(\omega_{qrz})$*:*

- $\tau_{misc}^{cst}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{x}\}$ *denotes a value constant* x
- $\tau_{misc}^{id}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{y}\}$ *denotes an identifier* y
- $\tau_{misc}^{br}(\omega_{qrz}) \stackrel{def}{=} \{(\tau(\omega_{qrz}))\}$ *denotes a bracket*
- $\tau_{misc}^{true}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{true}\}$ *denotes a* `true`*-constant*
- $\tau_{misc}^{false}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{false}\}$ *denotes a* `false`*-constant*
- $\tau_{misc}^{arr}(\omega_{qrz}) \stackrel{def}{=} \{x\texttt{[}n\texttt{]}\}$ *denotes access to index* n *of array* x

Syntax of comparison operators $\tau_{comp}(\omega_{qrz}) \in \mathcal{T}_{comp}(\omega_{qrz})$*:*

- $\tau_{comp}^{eq}(\omega_{qrz}) \stackrel{def}{=} \{\pi_1\ \texttt{==}\ \pi_2\}$ *denotes equality*

- $\tau_{comp}^{ne}(\omega_{qrz}) \stackrel{def}{=} \{\pi_1 \texttt{ != } \pi_2\}$ *denotes inequality*
- $\tau_{comp}^{gt}(\omega_{qrz}) \stackrel{def}{=} \{\pi_1 \texttt{ > } \pi_2\}$ *denotes greater than*
- $\tau_{comp}^{ge}(\omega_{qrz}) \stackrel{def}{=} \{\pi_1 \texttt{ >= } \pi_2\}$ *denotes greater than/equal to*
- $\tau_{comp}^{lt}(\omega_{qrz}) \stackrel{def}{=} \{\pi_1 \texttt{ < } \pi_2\}$ *denotes lower than*
- $\tau_{comp}^{le}(\omega_{qrz}) \stackrel{def}{=} \{\pi_1 \texttt{ <= } \pi_2\}$ *denotes lower than/equal to*

Syntax of arithmetic operators $\tau_{arith}(\omega_{qrz}) \in \mathcal{T}_{arith}(\omega_{qrz})$*:*

- $\tau_{arith}^{mul}(\omega_{qrz}) \stackrel{def}{=} \{\eta_1 \texttt{ * } \eta_2\}$ *denotes a multiplication*
- $\tau_{arith}^{div}(\omega_{qrz}) \stackrel{def}{=} \{\eta_1 \texttt{ / } \eta_2\}$ *denotes a division*
- $\tau_{arith}^{add}(\omega_{qrz}) \stackrel{def}{=} \{\eta_1 \texttt{ + } \eta_2\}$ *denotes an addition*
- $\tau_{arith}^{sub}(\omega_{qrz}) \stackrel{def}{=} \{\eta_1 \texttt{ - } \eta_2\}$ *denotes a subtraction*
- $\tau_{arith}^{expt}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{exp(}\eta_{r,1}\texttt{,}\eta_2\texttt{)}\}$ *denotes an exponential function*
- $\tau_{arith}^{mod}(\omega_{qrz}) \stackrel{def}{=} \{\eta_{i,1} \texttt{ \% } \eta_{i,2}\}$ *denotes a modulo function*
- $\tau_{arith}^{um}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{-}\eta_1\}$ *denotes an unary minus*

Syntax of bitwise operators $\tau_{bv}(\omega_{qrz}) \in \mathcal{T}_{bv}(\omega_{qrz})$*:*

- $\tau_{bv}^{and}(\omega_{qrz}) \stackrel{def}{=} \{\lambda_1 \texttt{ \& } \ldots \texttt{ \& } \lambda_n\}$ *denotes a conjunction*
- $\tau_{bv}^{or}(\omega_{qrz}) \stackrel{def}{=} \{\lambda_1 \texttt{ | } \ldots \texttt{ | } \lambda_n\}$ *denotes a disjunction*
- $\tau_{bv}^{xor}(\omega_{qrz}) \stackrel{def}{=} \{\lambda_1 \texttt{ ^ } \lambda_2\}$ *denotes an exclusive or*
- $\tau_{bv}^{not}(\omega_{qrz}) \stackrel{def}{=} \{\texttt{!}\lambda_1\}$ *denotes a negation*

Syntax of conditional operators $\tau_{cond}(\omega_{qrz}) \in \mathcal{T}_{cond}(\omega_{qrz})$*:*

- $\tau_{cond}^{sel}(\omega_{qrz}) \stackrel{def}{=} \left\{\lambda^b\texttt{?}\pi_1\texttt{:}\pi_2\texttt{)}\right\}$ *denotes a conditional operator*

Definition A.12 (Type system of expressions in *Quartz* models). *The type system of* $\tau(\omega_{qrz})$ *is defined as follows [Sch09]:*

Type system of comparison operators $\tau_{comp}^{\gamma}(\omega_{qrz}) \in \mathcal{T}_{comp}(\omega_{qrz})$*, considering* $\gamma \in \{eq, ne, gt, ge, lt, le\}$*:*

- $$\frac{\pi_1 : \alpha(\omega_{qrz}) \quad \pi_2 : \alpha(\omega_{qrz})}{\tau_{comp}^{\gamma}(\omega_{qrz}) : \alpha_{bv}^{bool}(\omega_{qrz})}$$

Type system of arithmetic operators $\tau_{arith}^{\gamma}(\omega_{qrz}) \in \mathcal{T}_{arith}(\omega_{qrz})$*, considering* $\gamma \in \{mul, div, add, sub, um\}$*:*

- $$\frac{\eta_1 : \alpha_{num}(\omega_{qrz}) \quad [\eta_2 : \alpha_{num}(\omega_{qrz})]}{\tau_{arith}^{\gamma}(\omega_{qrz}) : \alpha_{num}(\omega_{qrz})}$$

Type system of modulo operator $\tau^{mod}_{arith}(\omega_{qrz}) \in \mathcal{T}_{arith}(\omega_{qrz})$*:*

- $$\frac{\eta^i_1 : \alpha_i(\omega_{qrz}) \quad \eta^i_2 : \alpha_i(\omega_{qrz})}{\tau^{mod}_{arith}(\omega_{qrz}) : \alpha_i(\omega_{qrz})}$$

Type system of exponential function operator $\tau^{expt}_{arith}(\omega_{qrz}) \in \mathcal{T}_{arith}(\omega_{qrz})$*:*

- $$\frac{\eta^r_1 : \alpha_r(\omega_{qrz}) \quad \eta_2 : \alpha_{num}(\omega_{qrz})}{\tau^{expt}_{arith}(\omega_{qrz}) : \alpha_r(\omega_{qrz})}$$

Type system of bitwise operators $\tau^{\gamma}_{bv}(\omega_{qrz}) \in \mathcal{T}_{bv}(\omega_{qrz})$*, considering* $\gamma \in \{and, or, xor, not\}$*:*

- $$\frac{\lambda_1 : \alpha_{bv}(\omega_{qrz}) \quad [\lambda_2 : \alpha_{bv}(\omega_{qrz}) \quad \cdots \quad \lambda_n : \alpha_{bv}(\omega_{qrz})]}{\tau^{\gamma}_{bv}(\omega_{qrz}) : \alpha^{bool}_{bv}(\omega_{qrz})}$$

Type system of conditional operator $\tau^{sel}_{cond}(\omega_{qrz}) \in \mathcal{T}_{cond}(\omega_{qrz})$*:*

- $$\frac{\lambda^b : \alpha^{bool}_{bv}(\omega_{qrz}) \quad \pi_1 : \alpha(\omega_{qrz}) \quad \pi_2 : \alpha(\omega_{qrz})}{\tau^{sel}_{cond}(\omega_{qrz}) : \alpha(\omega_{qrz})}$$

Definition A.13 (Semantics of expressions in *Quartz* models). *The semantics of* $\tau(\omega_{qrz}) \in \mathcal{T}(\omega_{qrz})$ *are defined as follows [Sch09]:*

Semantics of constants and general expressions $\tau_{misc}(\omega_{qrz}) \in \mathcal{T}_{misc}(\omega_{qrz})$*:*

- $[\![\tau^{cst}_{misc}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![x]\!]_\xi$
- $[\![\tau^{id}_{misc}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} y$
- $[\![\tau^{br}_{misc}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} (\tau_{misc}(\omega_{qrz}))$
- $[\![\tau^{true}_{misc}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} true$
- $[\![\tau^{false}_{misc}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} false$
- $[\![\tau^{arr}_{misc}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![x\,[n]\,]\!]_\xi$

Semantics of comparison operators $\tau_{comp}(\omega_{qrz}) \in \mathcal{T}_{comp}(\omega_{qrz})$*:*

- $[\![\tau^{eq}_{comp}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\pi_1]\!]_\xi = [\![\pi_2]\!]_\xi$
- $[\![\tau^{ne}_{comp}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\pi_1]\!]_\xi \neq [\![\pi_2]\!]_\xi$
- $[\![\tau^{gt}_{comp}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\pi_1]\!]_\xi > [\![\pi_2]\!]_\xi$
- $[\![\tau^{ge}_{comp}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\pi_1]\!]_\xi >= [\![\pi_2]\!]_\xi$
- $[\![\tau^{lt}_{comp}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\pi_1]\!]_\xi < [\![\pi_2]\!]_\xi$
- $[\![\tau^{le}_{comp}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\pi_1]\!]_\xi <= [\![\pi_2]\!]_\xi$

Semantics of arithmetic operators $\tau_{arith}(\omega_{qrz}) \in \mathcal{T}_{arith}(\omega_{qrz})$*:*

- $[\![\tau^{mul}_{arith}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\eta_1]\!]_\xi \cdot [\![\eta_2]\!]_\xi$
- $[\![\tau^{div}_{arith}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} \frac{[\![\eta_1]\!]_\xi}{[\![\eta_2]\!]_\xi}$
- $[\![\tau^{add}_{arith}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\eta_1]\!]_\xi + [\![\eta_2]\!]_\xi$
- $[\![\tau^{sub}_{arith}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\eta_1]\!]_\xi - [\![\eta_2]\!]_\xi$
- $[\![\tau^{expt}_{arith}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\eta^r_1]\!]^{[\![\eta_2]\!]_\xi}_\xi$
- $[\![\tau^{mod}_{arith}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\eta^i_1]\!]_\xi \ mod \ [\![\eta^i_2]\!]_\xi$
- $[\![\tau^{um}_{arith}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} -[\![\eta_1]\!]_\xi$

Semantics of Boolean operators $\tau_{bv}(\omega_{qrz}) \in \mathcal{T}_{bv}(\omega_{qrz})$*:*

- $[\![\tau^{and}_{bv}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\lambda_1]\!]_\xi \wedge \cdots \wedge [\![\lambda_n]\!]_\xi$
- $[\![\tau^{or}_{bv}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\lambda_1]\!]_\xi \vee \cdots \vee [\![\lambda_n]\!]_\xi$
- $[\![\tau^{xor}_{bv}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} [\![\lambda_1]\!]_\xi \oplus [\![\lambda_2]\!]_\xi$
- $[\![\tau^{not}_{bv}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} \neg[\![\lambda_1]\!]_\xi$

Semantics of conditional operators $\tau^{sel}_{cond}(\omega_{qrz}) \in \mathcal{T}_{cond}(\omega_{qrz})$*:*

- $[\![\tau^{sel}_{cond}(\omega_{qrz})]\!]_\xi \stackrel{def}{=} \begin{cases} [\![\pi_1]\!]_\xi, & \text{if } [\![\lambda^b]\!]_\xi = true \\ [\![\pi_2]\!]_\xi, & otherwise \end{cases}$

Definition A.14 (SOS transition rules of expressions in *Quartz* models). *The SOS transition rules of* $\tau(\omega_{qrz}) \in \mathcal{T}(\omega_{qrz})$ *are defined as follows [Sch09]:*

- $\langle \xi, \tau(\omega_{qrz})\rangle \xrightarrow{\mathbb{T}} \langle nothing, \{[\![\tau(\omega_{qrz})]\!]_\xi\}, true\rangle$

A.2.7. Abortions in *Quartz* Models

Definition A.15 (Syntax of abortions in *Quartz* models). *The syntax of strong delayed abortions* $\Sigma^{reg}_{abort}(\omega_{qrz})$ *and strong immediate abortions* $\Sigma^{imm}_{abort}(\omega_{qrz})$ *is defined as follows [Sch09]:*

- $\sigma^{reg}_{abort}(\omega_{qrz}) \stackrel{def}{=}$ `abort` $\Sigma(\omega_{qrz})$ `when(`$\lambda^b(\omega_{qrz})$`);` *denotes a strong delayed abortion*
- $\sigma^{imm}_{abort}(\omega_{qrz}) \stackrel{def}{=}$ `immediate abort` $\Sigma(\omega_{qrz})$ `when(`$\lambda^b(\omega_{qrz})$`);` *denotes a strong immediate abortion*

Definition A.16 (SOS transition rules of abortions in *Quartz* models). *The SOS transition rules of strong delayed abortions* $\Sigma^{reg}_{abort}(\omega_{qrz})$ *and strong immediate abortions* $\Sigma^{imm}_{abort}(\omega_{qrz})$ *are defined as follows [Sch09]:*

SOS transition rules of $\sigma^{reg}_{abort}(\omega_{qrz}) \in \Sigma^{reg}_{abort}(\omega_{qrz})$*:*

- $$\frac{\langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}, true \rangle}{\langle \xi, \{\texttt{abort } \Sigma \texttt{ when}(\lambda^b);\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}, true \rangle}$$
- $$\frac{\langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle \Sigma', \mathcal{D}, false \rangle}{\langle \xi, \{\texttt{abort } \Sigma \texttt{ when}(\lambda^b);\} \rangle \xrightarrow{\mathbb{T}} \langle \left\{ \begin{array}{l} \texttt{immediate abort } \Sigma' \\ \texttt{when}(\lambda^b); \end{array} \right\}, \mathcal{D}, false \rangle}$$

SOS transition rules of $\sigma^{imm}_{abort}(\omega_{qrz}) \in \Sigma^{imm}_{abort}(\omega_{qrz})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true}{\langle \xi, \left\{ \begin{array}{l} \texttt{immediate abort } \Sigma \\ \texttt{when}(\lambda^b); \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{immediate abort } \Sigma \\ \texttt{when}(\lambda^b); \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}, true \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle \Sigma', \mathcal{D}, false \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{immediate abort } \Sigma \\ \texttt{when}(\lambda^b); \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \left\{ \begin{array}{l} \texttt{immediate abort } \Sigma' \\ \texttt{when}(\lambda^b); \end{array} \right\}, \mathcal{D}, false \rangle}$$

A.2.8. Assignments in *Quartz* Models

Definition A.17 (Syntax of assignments in *Quartz* models). *The syntax of* $\sigma^{imm}_{ass}(\omega_{qrz}) \in \Sigma^{imm}_{ass}(\omega_{qrz})$ *and* $\sigma^{del}_{ass}(\omega_{qrz}) \in \Sigma^{del}_{ass}(\omega_{qrz})$ *is defined as follows [Sch09]:*

- $\sigma^{imm}_{ass}(\omega_{qrz}) \stackrel{def}{=} \{\ x = \tau;\ \}$
  - $\texttt{emit(x)} :\equiv \sigma^{imm}_{ass}(\omega_{qrz}) \iff \tau = true$
- $\sigma^{del}_{ass}(\omega_{qrz}) \stackrel{def}{=} \{\ \texttt{next}(x) = \tau(x);\ \}$

Definition A.18 (SOS transition rules of assignments in *Quartz* models). *The SOS transition rules of* $\sigma^{imm}_{ass}(\omega_{qrz}) \in \Sigma^{imm}_{ass}(\omega_{qrz})$ *and* $\sigma^{del}_{ass}(\omega_{qrz}) \in \Sigma^{del}_{ass}(\omega_{qrz})$ *are defined as follows [Sch09]:*

- $\langle \xi, \sigma^{imm}_{ass} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{[\![x]\!]_\xi = [\![\tau]\!]_\xi\}, true \rangle$
- $\langle \xi, \sigma^{del}_{ass} \rangle \xrightarrow{\mathbb{T}} \langle \{x = [\![\tau(x)]\!]_\xi\}, [\![\tau(x)]\!]_\xi, true \rangle$

A.2.9. Await Statements in *Quartz* Models

Definition A.19 (Syntax and SOS transition rules of await statements in *Quartz* models). *The syntax of strong delayed await statements* $\Sigma^{reg}_{ass}(\omega_{qrz})$ *and immediate await statements* $\Sigma^{imm}_{ass}(\omega_{qrz})$ *is defined as follows [Sch09]. The SOS transitions rules are derived from equivalent constructs:*

- $\sigma^{reg}_{ass}(\omega_{qrz}) \stackrel{def}{=}$ `await(`$\lambda^b(\omega_{qrz})$`);` *denotes a strong delayed await statements*
  - $\sigma^{reg}_{ass}(\omega_{qrz}) \coloneqq$ `do pause; while(!`$\lambda^b(\omega_{qrz})$`);`
  - $\sigma^{reg}_{ass}(\omega_{qrz}) \coloneqq$ `abort halt; when(`$\lambda^b(\omega_{qrz})$`);`
- $\sigma^{imm}_{ass}(\omega_{qrz}) \stackrel{def}{=}$ `immediate await(`$\lambda^b(\omega_{qrz})$`);` *denotes an immediate await statements*
  - $\sigma^{imm}_{ass}(\omega_{qrz}) \coloneqq$ `while(!`$\lambda^b(\omega_{qrz})$`) pause;`
  - $\sigma^{imm}_{ass}(\omega_{qrz}) \coloneqq$ `immediate abort halt; when(`$\lambda^b(\omega_{qrz})$`);`

A.2.10. Synchronous Concurrency in *Quartz* Models

Definition A.20 (Syntax of synchronous concurrency in *Quartz* models). *The syntax of a synchronous parallel statement* $\Sigma_{conc}(\omega_{qrz})$ *is defined as follows [Sch09]:*

- $\Sigma_{conc}(\omega_{qrz}) \stackrel{def}{=} \Sigma_1(\omega_{qrz}) \parallel \Sigma_2(\omega_{qrz})$

Definition A.21 (SOS transition rules of synchronous concurrency in *Quartz* models). *The SOS transition rules of a synchronous parallel statement* $\Sigma_{conc}(\omega_{qrz})$ *are defined as follows [Sch09]:*

- $$\frac{\langle \xi, \Sigma_1 \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle \wedge \langle \xi, \Sigma_2 \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \{\Sigma_1 \parallel \Sigma_2\} \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \{\Sigma_1' \parallel \Sigma_2'\}, \mathcal{D}_1 \bigcup \mathcal{D}_2, f_1 \wedge f_2 \rangle}$$

A.2.11. Conditions in *Quartz* Models

Definition A.22 (Syntax of conditions in *Quartz* models). *The syntax of* $\sigma^{it}_{cond}(\omega^{\varphi}_{st}) \in \Sigma^{it}_{cond}(\omega^{\varphi}_{st})$ *and* $\sigma^{ite}_{cond}(\omega^{\varphi}_{st}) \in \Sigma^{ite}_{cond}(\omega^{\varphi}_{st})$ *is defined as follows [Sch09]:*

- $\sigma^{it}_{cond}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{if(}\lambda^b(\sigma^{it}_{cond}(\omega_{qrz}))\texttt{)\{} \\ \quad \Sigma_1(\sigma^{it}_{cond}(\omega_{qrz})) \\ \texttt{\}} \end{array} \right\}$
- $\sigma^{ite}_{cond}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{if(}\lambda^b(\sigma^{ite}_{cond}(\omega_{qrz}))\texttt{)\{} \\ \quad \Sigma_1(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{\}else\{} \\ \quad \Sigma_2(\sigma^{ite}_{cond}(\omega_{qrz})) \\ \texttt{\}} \end{array} \right\}$

Definition A.23 (SOS transition rules of conditions in *Quartz* models). *The SOS transition rules of* $\sigma^{it}_{cond}(\omega_{qrz}) \in \Sigma^{it}_{cond}(\omega_{qrz})$ *and* $\sigma^{ite}_{cond}(\omega_{qrz}) \in \Sigma^{ite}_{cond}(\omega_{qrz})$ *are defined as follows [Sch09]:*

SOS transition rules of $\sigma^{it}_{cond}(\omega_{qrz}) \in \Sigma^{it}_{cond}(\omega_{qrz})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}{\langle \xi, \{\texttt{if}(\lambda^b)\{\Sigma_1\}\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}{\langle \xi, \{\texttt{if}(\lambda^b)\{\Sigma_1\}\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}$$

SOS transition rules of $\sigma^{ite}_{cond}(\omega_{qrz}) \in \Sigma^{ite}_{cond}(\omega_{qrz})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{if}(\lambda^b)\{\Sigma_1 \\ \}\texttt{else}\{\Sigma_2\} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{if}(\lambda^b)\{\Sigma_1 \\ \}\texttt{else}\{\Sigma_2\} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}$$

A.2.12. Halt Statements in *Quartz* Models

Definition A.24 (Syntax and SOS transitions rules of halt statements in *Quartz* models). *The syntax of halt statements* $\Sigma_{halt}(\omega_{qrz})$ *is defined as follows [Sch09]. The SOS transitions rules are derived from equivalent constructs:*

- $\sigma_{halt}(\omega_{qrz}) \stackrel{def}{=}$ `halt;` *denotes an infinite loop doing nothing*
  - $\sigma_{halt}(\omega_{qrz}) :\equiv$ `do pause; while(true);`

A.2.13. Module Invocations in *Quartz* Models

Definition A.25 (Syntax of *Quartz* module invocations). *The syntax of* Quartz *module invocations of a non instantiated module* $\sigma^{f'}_{inv}(\omega_{qrz}) \in \Sigma^{f'}_{inv}(\omega_{qrz})$ *and of an instantiated module* $\sigma^{fb}_{inv}(\omega_{qrz}) \in \Sigma^{fb}_{inv}(\omega_{qrz})$ *is defined as follows, where k represents the module and x the instance name:*

- $\sigma^{f'}_{inv}(\omega_{qrz}) \stackrel{def}{=} \left\{ k(\mathcal{I}(\sigma^{f'}_{inv}(\omega_{qrz})), \mathcal{O}(\sigma^{f'}_{inv}(\omega_{qrz})))[;] \right\}$ *denotes a* Quartz *module invocations of a non instantiated module*
- $\sigma^{fb}_{inv}(\omega_{qrz}) \stackrel{def}{=} \left\{ x : k(\mathcal{I}(\sigma^{fb}_{inv}(\omega_{qrz})), \mathcal{O}(\sigma^{fb}_{inv}(\omega_{qrz})))[;] \right\}$ *denotes a* Quartz *module invocations of an instantiated module*

Definition A.26 (SOS transition rules of *Quartz* module invocations). *The SOS transition rules of a non instantiated module* $\sigma^{f'}_{inv}(\omega_{qrz}) \in \Sigma^{f'}_{inv}(\omega_{qrz})$ *and of an instantiated module* $\sigma^{fb}_{inv}(\omega_{qrz}) \in \Sigma^{fb}_{inv}(\omega_{qrz})$ *is defined as follows, where k represents the module and x the instance name:*

SOS transition rules of $\sigma^{f'}_{inv}(\omega_{qrz}) \in \Sigma^{f'}_{inv}(\omega_{qrz})$*:*

- $\langle \xi, \sigma_{inv}^{f'} \rangle \xrightarrow{\mathbb{T}} \langle \sigma_{inv}^{f''}, \mathcal{D}, f \rangle$

SOS transition rules of $\sigma_{inv}^{fb}(\omega_{qrz}) \in \Sigma_{inv}^{fb}(\omega_{qrz})$*:*

- $\langle \xi, \sigma_{inv}^{fb} \rangle \xrightarrow{\mathbb{T}} \langle \sigma_{inv}^{fb'}, \mathcal{D}, f \rangle$

A.2.14. Loops in *Quartz* Models

Definition A.27 (Syntax of loops in *Quartz* models). *The syntax of* $\sigma_{loop}^{foot}(\omega_{qrz}) \in \Sigma_{loop}^{foot}(\omega_{qrz})$, $\sigma_{loop}^{head}(\omega_{qrz}) \in \Sigma_{loop}^{head}(\omega_{qrz})$, *and* $\sigma_{loop}^{inf}(\omega_{qrz}) \in \Sigma_{loop}^{inf}(\omega_{qrz})$ *is defined as follows:*

- $\sigma_{loop}^{foot}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{do:} \\ \quad \Sigma(\sigma_{loop}^{foot}(\omega_{qrz})) \\ \texttt{while}(\lambda^b(\sigma_{loop}^{foot}(\omega_{qrz}))); \end{array} \right\}$
- $\sigma_{loop}^{head}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{while}(\lambda^b(\sigma_{loop}^{head}(\omega_{qrz}))) \{ \\ \quad \Sigma(\sigma_{loop}^{head}(\omega_{qrz})) \\ \} \end{array} \right\}$
- $\sigma_{loop}^{inf}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{loop} \{ \\ \quad \Sigma(\sigma_{loop}^{inf}(\omega_{qrz})) \\ \} \end{array} \right\}$
  - $\sigma_{loop}^{inf}(\omega_{qrz}) :\equiv \sigma_{loop}^{foot}(\omega_{qrz}) \iff \lambda^b(\sigma_{loop}^{foot}(\omega_{qrz})) = true$

Definition A.28 (SOS transition rules of loops in *Quartz* models). *The SOS transition rules of* $\sigma_{loop}^{foot}(\omega_{qrz}) \in \Sigma_{loop}^{foot}(\omega_{qrz})$ *and* $\sigma_{loop}^{head}(\omega_{qrz}) \in \Sigma_{loop}^{head}(\omega_{qrz})$ *are defined as follows:*

SOS transition rules of $\sigma_{loop}^{foot}(\omega_{qrz}) \in \Sigma_{loop}^{foot}(\omega_{qrz})$*:*

- $$\frac{\langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle \Sigma', \mathcal{D}, false \rangle}{\langle \xi, \{\texttt{do } \Sigma \texttt{ while}(\lambda^b);\} \rangle \xrightarrow{\mathbb{T}} \langle \{\Sigma';\ \texttt{while}(\lambda^b)\ \Sigma\}, \mathcal{D}, false \rangle}$$

SOS transition rules of $\sigma_{loop}^{head}(\omega_{qrz}) \in \Sigma_{loop}^{head}(\omega_{qrz})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle \Sigma', \mathcal{D}, false \rangle}{\langle \xi, \{\texttt{while}(\lambda^b)\ \Sigma\} \rangle \xrightarrow{\mathbb{T}} \langle \{\Sigma';\ \texttt{while}(\lambda^b)\ \Sigma\}, \mathcal{D}, false \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false}{\langle \xi, \{\texttt{while}(\lambda^b)\ \Sigma\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}$$

A.2.15. Nothing Statements in *Quartz* Models

Definition A.29 (Syntax of nothing statements in *Quartz* models). *The syntax of nothing statements $\Sigma_{nothing}(\omega_{qrz})$ is defined as follows [Sch09]:*

- $\Sigma_{nothing}(\omega_{qrz}) \stackrel{def}{=}$ `nothing;`

Definition A.30 (SOS transition rules of nothing statements in *Quartz* models). *The SOS transition rules of nothing statements $\Sigma_{nothing}(\omega_{qrz})$ are defined as follows [Sch09]:*

- $\langle \xi, \texttt{nothing;} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle$

A.2.16. Pause Statements in *Quartz* Models

Definition A.31 (Syntax of pause statements in *Quartz* models). *The syntax of pause statements $\Sigma_{pause}(\omega_{qrz})$ is defined as follows [Sch09]:*

- $\Sigma_{pause}(\omega_{qrz}) \stackrel{def}{=}$ `pause;`

Definition A.32 (SOS transition rules of pause statements in *Quartz* models). *The SOS transition rules of pause statements $\Sigma_{pause}(\omega_{qrz})$ are defined as follows [Sch09]:*

- $\langle \xi, \texttt{pause;} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, false \rangle$

A.2.17. Sequences in *Quartz* Models

Definition A.33 (Syntax of sequences in SCL models). *The syntax of a sequence $\Sigma_{seq}(\omega_{qrz})$ is defined as follows [Sch09]:*

- $\Sigma_{seq}(\omega_{qrz}) \stackrel{def}{=} \left\{ \begin{array}{c} \sigma_1(\omega_{qrz}); \\ \sigma_2(\omega_{qrz}); \\ \vdots \\ \sigma_n(\omega_{qrz}); \end{array} \right\}$

Definition A.34 (SOS transition rules of sequences in *Quartz* models). *The SOS transition rules of a sequence $\Sigma_{seq}(\omega_{qrz})$ are defined as follows [Sch09]:*

- $$\frac{\langle \xi, \sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \sigma_1', \mathcal{D}_1, false \rangle}{\langle \xi, \{\sigma_1; \sigma_2;\} \rangle \xrightarrow{\mathbb{T}} \langle \{\sigma_1'; \sigma_2;\}, \mathcal{D}_1, false \rangle}$$
- $$\frac{\langle \xi, \sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \sigma_1', \mathcal{D}_1, true \rangle \wedge \langle \xi, \sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \{\sigma_1; \sigma_2;\} \rangle \xrightarrow{\mathbb{T}} \langle \sigma_2', \{\mathcal{D}_1 \bigcup \mathcal{D}_2\}, f_2 \rangle}$$

A.3. SCL Models

A.3.1. SCL Variants and Declaration

Definition A.35 (Syntax of SCL elements). *An SCL module is declared as follows, assuming fixed order of* $\Delta_{idcl}(\omega_{scl})$, $\Delta_{vdcl}(\omega_{scl})$, *and* $\Sigma(\omega_{scl})$*:*

- $\delta_\omega(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{module } a_n(\omega_{scl})\{ \\ \quad [\Delta_{idcl}(\omega_{scl})] \quad [\Delta_{vdcl}(\omega_{scl})] \quad [\Sigma(\omega_{scl})] \\ \} \end{array} \right\}$

Definition A.36 (Semantics of SCL elements). *The semantics of* ω_{scl} *are defined as follows:*

- $[\![\delta_\omega(\omega_{scl})]\!]_\xi \stackrel{def}{=}$ { *defines an SCL model element* ω_{scl} *with* $\Sigma(\omega_{scl})$*, started at time* t *with initial conditions set by* $\Delta_{idcl}(\omega_{scl})$ *and* $\Delta_{vdcl}(\omega_{scl})$ *when invoked, terminating at time* $t + \theta$ *and preserving internal state across* macro steps.}

A.3.2. SCL Interfaces

Definition A.37 (Syntax of SCL interfaces). *The syntax of* $\Delta_{in}(\omega_{scl})$, $\Delta_{out}(\omega_{scl})$, *and* $\Delta_{inout}(\omega_{scl})$ *is defined as follows:*

- $\Delta_{in}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{input [signal]} \ \alpha_1^{[+]} \ x_1\texttt{[=}w_1\texttt{][;]} \\ \vdots \\ \texttt{input [signal]} \ \alpha_n^{[+]} \ x_n\texttt{[=}w_n\texttt{][;]} \end{array} \right\}$
- $\Delta_{out}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{output [signal]} \ \alpha_1^{[+]} \ x_1\texttt{[=}w_1\texttt{][;]} \\ \vdots \\ \texttt{output [signal]} \ \alpha_n^{[+]} \ x_n\texttt{[=}w_n\texttt{][;]} \end{array} \right\}$
- $\Delta_{inout}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{input output} \ \alpha_1^{[+]} \ x_1\texttt{[=}w_1\texttt{][;]} \\ \vdots \\ \texttt{input output} \ \alpha_n^{[+]} \ x_n\texttt{[=}w_n\texttt{][;]} \end{array} \right\}$

Definition A.38 (Semantics of SCL interfaces). *The semantics of* $\Delta_{in}(\omega_{scl})$, $\Delta_{out}(\omega_{scl})$, *and* $\Delta_{inout}(\omega_{scl})$ *are defined as follows:*

- $[\![\Delta_{in}(\omega_{scl})]\!]_\xi \stackrel{def}{=}$ { *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$*. These variables are assigned to optional predefined default values* $[\![w_1]\!]_\xi, \ldots, [\![w_n]\!]_\xi$ *or to externally supplied values at time* t *when* ω_{scl} *is invoked. Their values can be modified until* ω_{scl} *terminates at time* $t+\theta$*, i.e., all instructions of* ω_{scl} *have been processed. Variables classified as* signal *variables are reset to their default values if they are not assigned in the current* macro step. }

- $[\![\Delta_{out}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \{$ *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. *These variables are assigned to optional predefined default values* $[\![w_1]\!]_\xi, \ldots, [\![w_n]\!]_\xi$ *at time* t *when* ω_{scl} *is invoked, and passed to invoking element in each* macro step *with current value (assumed SCL was synthesized into an SCChart that is instantiated by another model). Variables classified as* signal *variables are reset to their default values if they are not assigned in the current* macro step. $\}$

- $[\![\Delta_{inout}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \{$ *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. *These variables are assigned to externally supplied values at time* t *when* ω_{scl} *is invoked. Their values can be modified until* ω_{scl} *terminates at time* $t+\theta$, *and passed to invoking element in each* macro step *with current value (assumed SCL was synthesized into an SCChart that is instantiated by another model).* $\}$

A.3.3. Local Variables in SCL Models

Definition A.39 (Syntax of local variables in SCL models). *The syntax of* $\Delta_{local}(\omega_{scl})$ *is defined as follows:*

- $\Delta_{local}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \alpha_1^{[+]}\ x_1\texttt{[=}w_1\texttt{][;]} \\ \vdots \\ \alpha_n^{[+]}\ x_n\texttt{[=}w_n\texttt{][;]} \end{array} \right\}$

Definition A.40 (Semantics of local variables in SCL models). *The semantics of* $\Delta_{local}(\omega_{scl})$ *are defined as follows:*

- $[\![\Delta_{local}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \{$ *defines a set of variables* $x_1, \ldots, x_n$ *with corresponding data types* $[\![\alpha_1^{[+]}]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. *These variables are assigned to optional predefined default values* $[\![w_1]\!]_\xi, \ldots, [\![w_n]\!]_\xi$ *at time* t *when* ω_{scl} *is invoked and keep their values when switching from one* macro step *to another. These variables can be modified and processed locally until* ω_{scl} *terminates at time* $t+\theta$, *i.e., all instructions of* ω_{scl} *have been processed.* $\}$

A.3.4. Elementary SCL Data Types and Fields

Definition A.41 (Syntax of elementary SCL data types and fields). *The syntax of* $\alpha(\omega_{scl}) \in \mathcal{A}(\omega_{scl})$ *and* $\alpha^+(\omega_{scl}) \in \mathcal{A}^+(\omega_{scl})$ *is defined as follows:*

Syntax of bit vector data types $\alpha_{bv}(\omega_{scl}) \in \mathcal{A}_{bv}(\omega_{scl})$*:*

- $\alpha_{bv}^{bool}(\omega_{scl}) \stackrel{def}{=} \{\texttt{bool}\}$ *denotes Boolean values*

Syntax of integer data types $\alpha_i(\omega_{scl}) \in \mathcal{A}_i(\omega_{scl})$*:*

- $\alpha_i^{int}(\omega_{scl}) \stackrel{def}{=} \{\texttt{int}\}$ *denotes bounded signed integers*

Syntax of floating-point data types $\alpha_r(\omega_{scl}) \in \mathcal{A}_r(\omega_{scl})$*:*

- $\alpha_r^{real}(\omega_{scl}) \stackrel{def}{=} \{\texttt{float}\}$ *denotes floating-point values*

Syntax of numeric data types $\alpha_{num}(\omega_{scl}) \in \mathcal{A}_i(\omega_{scl}) \bigcup \mathcal{A}_r(\omega_{scl})$*:*

- $\alpha_{num}(\omega_{scl}) \in \{\alpha_i(\omega_{scl}), \alpha_r(\omega_{scl})\}$

Syntax of duration data types $\alpha_{dur}(\omega_{scl}) \in \mathcal{A}_{dur}(\omega_{scl})$*:*

- $\alpha_{dur}^{time}(\omega_{scl}) \stackrel{def}{=} \{\texttt{int}\}$ *denotes interval value in bounded integer*

Syntax of data type fields $\alpha^+(\omega_{scl}) \in \mathcal{A}^+(\omega_{scl})$*:*

- $\alpha^+(\omega_{scl}) \stackrel{def}{=} \{\alpha(\omega_{scl})\ \ y\texttt{[n]}\}$ *denotes arrays*

Definition A.42 (Semantics of elementary SCL data types and fields). *The semantics of* $\alpha(\omega_{scl}) \in \mathcal{A}(\omega_{scl})$ *and* $\alpha^+(\omega_{scl}) \in \mathcal{A}^+(\omega_{scl})$ *are defined as follows:*

Semantics of bit vector data types $\alpha_{bv}(\omega_{scl}) \in \mathcal{A}_{bv}(\omega_{scl})$*:*

- $[\![\alpha_{bv}^{bool}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \langle\{false, true\}, false\rangle$

Semantics of integer data types $\alpha_i(\omega_{scl}) \in \mathcal{A}_i(\omega_{scl})$*:*

- $[\![\alpha_i^{int}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \langle\{-2^{31}+1, 2^{31}-1\}, 0\rangle$

Semantics of floating-point data types $\alpha_r(\omega_{scl}) \in \mathcal{A}_r(\omega_{scl})$*:*

- $[\![\alpha_r^{real}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \langle\{\textit{single precision floating-point (32 Bits)}\}, 0\rangle$

Semantics of data type fields $\alpha^+(\omega_{scl}) \in \mathcal{A}^+(\omega_{scl})$*:*

- $[\![\alpha^+(\omega_{scl})]\!]_\xi \stackrel{def}{=} \{$ *group with n elements of data type* $\alpha(\omega_{scl})\}$

A.3.5. Expressions in SCL Models

Definition A.43 (Syntax of expressions in SCL models). *The syntax of* $\mathcal{T}_{misc}(\omega_{scl})$*,* $\mathcal{T}_{comp}(\omega_{scl})$*,* $\mathcal{T}_{arith}(\omega_{scl})$*,* $\mathcal{T}_{bv}(\omega_{scl})$*, and* $\mathcal{T}_{cond}(\omega_{scl})$ *is defined as follows:*

Syntax of constants and general expressions $\tau_{misc}(\omega_{scl}) \in \mathcal{T}_{misc}(\omega_{scl})$*:*

- $\tau_{misc}^{cst}(\omega_{scl}) \stackrel{def}{=} \{\texttt{x}\}$ *denotes a value constant x*
- $\tau_{misc}^{id}(\omega_{scl}) \stackrel{def}{=} \{\texttt{y}\}$ *denotes an identifier y*
- $\tau_{misc}^{br}(\omega_{scl}) \stackrel{def}{=} \{(\tau(\omega_{scl}))\}$ *denotes a bracket*
- $\tau_{misc}^{true}(\omega_{scl}) \stackrel{def}{=} \{\texttt{true}\}$ *denotes a* `true`*-constant*

- $\tau_{misc}^{false}(\omega_{scl}) \stackrel{def}{=} \{\texttt{false}\}$ *denotes a* `false`*-constant*
- $\tau_{misc}^{arr}(\omega_{scl}) \stackrel{def}{=} \{x\texttt{[}n\texttt{]}\}$ *denotes access to index* n *of array* x

Syntax of comparison operators $\tau_{comp}(\omega_{scl}) \in \mathcal{T}_{comp}(\omega_{scl})$*:*

- $\tau_{comp}^{eq}(\omega_{scl}) \stackrel{def}{=} \{\pi_1 \texttt{ == } \pi_2\}$ *denotes equality*
- $\tau_{comp}^{ne}(\omega_{scl}) \stackrel{def}{=} \{\pi_1 \texttt{ != } \pi_2\}$ *denotes inequality*
- $\tau_{comp}^{gt}(\omega_{scl}) \stackrel{def}{=} \{\pi_1 \texttt{ > } \pi_2\}$ *denotes greater than*
- $\tau_{comp}^{ge}(\omega_{scl}) \stackrel{def}{=} \{\pi_1 \texttt{ >= } \pi_2\}$ *denotes greater than/equal to*
- $\tau_{comp}^{lt}(\omega_{scl}) \stackrel{def}{=} \{\pi_1 \texttt{ < } \pi_2\}$ *denotes lower than*
- $\tau_{comp}^{le}(\omega_{scl}) \stackrel{def}{=} \{\pi_1 \texttt{ <= } \pi_2\}$ *denotes lower than/equal to*

Syntax of arithmetic operators $\tau_{arith}(\omega_{scl}) \in \mathcal{T}_{arith}(\omega_{scl})$*:*

- $\tau_{arith}^{mul}(\omega_{scl}) \stackrel{def}{=} \{\eta_1 \texttt{ * } \eta_2\}$ *denotes a multiplication*
- $\tau_{arith}^{div}(\omega_{scl}) \stackrel{def}{=} \{\eta_1 \texttt{ / } \eta_2\}$ *denotes a division*
- $\tau_{arith}^{add}(\omega_{scl}) \stackrel{def}{=} \{\eta_1 \texttt{ + } \eta_2\}$ *denotes an addition*
- $\tau_{arith}^{sub}(\omega_{scl}) \stackrel{def}{=} \{\eta_1 \texttt{ - } \eta_2\}$ *denotes a subtraction*
- $\tau_{arith}^{mod}(\omega_{scl}) \stackrel{def}{=} \{\eta_{i,1} \texttt{ \% } \eta_{i,2}\}$ *denotes a modulo function*
- $\tau_{arith}^{um}(\omega_{scl}) \stackrel{def}{=} \{\texttt{-}\eta_1\}$ *denotes an unary minus*

Syntax of bitwise operators $\tau_{bv}(\omega_{scl}) \in \mathcal{T}_{bv}(\omega_{scl})$*:*

- $\tau_{bv}^{and}(\omega_{scl}) \stackrel{def}{=} \{\lambda_1 \texttt{ \& } \ldots \texttt{ \& } \lambda_n\}$ *denotes a conjunction*
- $\tau_{bv}^{or}(\omega_{scl}) \stackrel{def}{=} \{\lambda_1 \texttt{ | } \ldots \texttt{ | } \lambda_n\}$ *denotes a disjunction*
- $\tau_{bv}^{xor}(\omega_{scl}) \stackrel{def}{=} \{\lambda_1 \texttt{ ^ } \lambda_2\}$ *denotes an exclusive or*
- $\tau_{bv}^{not}(\omega_{scl}) \stackrel{def}{=} \{\texttt{!}\lambda_1\}$ *denotes a negation*

Syntax of conditional operators $\tau_{cond}(\omega_{scl}) \in \mathcal{T}_{cond}(\omega_{scl})$*:*

- $\tau_{cond}^{sel}(\omega_{scl}) \stackrel{def}{=} \left\{\lambda^b ? \pi_1 : \pi_2)\right\}$ *denotes a conditional operator*

Definition A.44 (Type system of expressions in SCL models). *The type system of* $\tau(\omega_{scl})$ *is defined as follows:*

Type system of comparison operators $\tau_{comp}^{\gamma}(\omega_{scl}) \in \mathcal{T}_{comp}(\omega_{scl})$*, considering* $\gamma \in \{eq, ne, gt, ge, lt, le\}$*:*

- $$\frac{\pi_1 : \alpha(\omega_{scl}) \quad \pi_2 : \alpha(\omega_{scl})}{\tau_{comp}^{\gamma}(\omega_{scl}) : \alpha_{bv}^{bool}(\omega_{scl})}$$

Type system of arithmetic operators $\tau^{\gamma}_{arith}(\omega_{scl}) \in \mathcal{T}_{arith}(\omega_{scl})$*, considering* $\gamma \in \{mul, div, add, sub, um\}$*:*

- $$\frac{\eta_1 : \alpha_{num}(\omega_{scl}) \quad [\eta_2 : \alpha_{num}(\omega_{scl})]}{\tau^{\gamma}_{arith}(\omega_{scl}) : \alpha_{num}(\omega_{scl})}$$

Type system of modulo operator $\tau^{mod}_{arith}(\omega_{scl}) \in \mathcal{T}_{arith}(\omega_{scl})$*:*

- $$\frac{\eta^i_1 : \alpha_i(\omega_{scl}) \quad \eta^i_2 : \alpha_i(\omega_{scl})}{\tau^{mod}_{arith}(\omega_{scl}) : \alpha_i(\omega_{scl})}$$

Type system of bitwise operators $\tau^{\gamma}_{bv}(\omega_{scl}) \in \mathcal{T}_{bv}(\omega_{scl})$*, considering* $\gamma \in \{and, or, xor, not\}$*:*

- $$\frac{\lambda_1 : \alpha_{bv}(\omega_{scl}) \quad [\lambda_2 : \alpha_{bv}(\omega_{scl}) \quad \cdots \quad \lambda_n : \alpha_{bv}(\omega_{scl})]}{\tau^{\gamma}_{bv}(\omega_{scl}) : \alpha^{bool}_{bv}(\omega_{scl})}$$

Type system of conditional operator $\tau^{sel}_{cond}(\omega_{scl}) \in \mathcal{T}_{cond}(\omega_{scl})$*:*

- $$\frac{\lambda^b : \alpha^{bool}_{bv}(\omega_{scl}) \quad \pi_1 : \alpha(\omega_{scl}) \quad \pi_2 : \alpha(\omega_{scl})}{\tau^{sel}_{cond}(\omega_{scl}) : \alpha(\omega_{scl})}$$

Definition A.45 (Semantics of expressions in SCL models). *The semantics of* $\tau(\omega_{scl}) \in \mathcal{T}(\omega_{scl})$ *are defined as follows:*

Semantics of constants and general expressions $\tau_{misc}(\omega_{scl}) \in \mathcal{T}_{misc}(\omega_{scl})$*:*

- $[\![\tau^{cst}_{misc}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![x]\!]_{\xi}$
- $[\![\tau^{id}_{misc}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} y$
- $[\![\tau^{br}_{misc}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} (\tau_{misc}(\omega_{scl}))$
- $[\![\tau^{true}_{misc}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} true$
- $[\![\tau^{false}_{misc}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} false$
- $[\![\tau^{arr}_{misc}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![x\,[n]\,]\!]_{\xi}$

Semantics of comparison operators $\tau_{comp}(\omega_{scl}) \in \mathcal{T}_{comp}(\omega_{scl})$*:*

- $[\![\tau^{eq}_{comp}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} = [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{ne}_{comp}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} \neq [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{gt}_{comp}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} > [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{ge}_{comp}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} >= [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{lt}_{comp}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} < [\![\pi_2]\!]_{\xi}$
- $[\![\tau^{le}_{comp}(\omega_{scl})]\!]_{\xi} \stackrel{def}{=} [\![\pi_1]\!]_{\xi} <= [\![\pi_2]\!]_{\xi}$

Semantics of arithmetic operators $\tau_{arith}(\omega_{scl}) \in \mathcal{T}_{arith}(\omega_{scl})$*:*

- $[\![\tau^{mul}_{arith}(\omega_{scl})]\!]_\xi \stackrel{def}{=} [\![\eta_1]\!]_\xi \cdot [\![\eta_2]\!]_\xi$
- $[\![\tau^{div}_{arith}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \frac{[\![\eta_1]\!]_\xi}{[\![\eta_2]\!]_\xi}$
- $[\![\tau^{add}_{arith}(\omega_{scl})]\!]_\xi \stackrel{def}{=} [\![\eta_1]\!]_\xi + [\![\eta_2]\!]_\xi$
- $[\![\tau^{sub}_{arith}(\omega_{scl})]\!]_\xi \stackrel{def}{=} [\![\eta_1]\!]_\xi - [\![\eta_2]\!]_\xi$
- $[\![\tau^{mod}_{arith}(\omega_{scl})]\!]_\xi \stackrel{def}{=} [\![\eta^i_1]\!]_\xi \ mod \ [\![\eta^i_2]\!]_\xi$
- $[\![\tau^{um}_{arith}(\omega_{scl})]\!]_\xi \stackrel{def}{=} -[\![\eta_1]\!]_\xi$

Semantics of Boolean operators $\tau_{bv}(\omega_{scl}) \in \mathcal{T}_{bv}(\omega_{scl})$*:*

- $[\![\tau^{and}_{bv}(\omega_{scl})]\!]_\xi \stackrel{def}{=} [\![\lambda_1]\!]_\xi \wedge \cdots \wedge [\![\lambda_n]\!]_\xi$
- $[\![\tau^{or}_{bv}(\omega_{scl})]\!]_\xi \stackrel{def}{=} [\![\lambda_1]\!]_\xi \vee \cdots \vee [\![\lambda_n]\!]_\xi$
- $[\![\tau^{xor}_{bv}(\omega_{scl})]\!]_\xi \stackrel{def}{=} [\![\lambda_1]\!]_\xi \oplus [\![\lambda_2]\!]_\xi$
- $[\![\tau^{not}_{bv}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \neg[\![\lambda_1]\!]_\xi$

Semantics of conditional operators $\tau^{sel}_{cond}(\omega_{scl}) \in \mathcal{T}_{cond}(\omega_{scl})$*:*

- $[\![\tau^{sel}_{cond}(\omega_{scl})]\!]_\xi \stackrel{def}{=} \begin{cases} [\![\pi_1]\!]_\xi, & \text{if } [\![\lambda^b]\!]_\xi = true \\ [\![\pi_2]\!]_\xi, & \text{otherwise} \end{cases}$

Definition A.46 (SOS transition rules of expressions in SCL models). *The SOS transition rules of* $\tau(\omega_{scl}) \in \mathcal{T}(\omega_{scl})$ *are defined as follows:*

- $\langle \xi, \tau(\omega_{scl}) \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{[\![\tau(\omega_{scl})]\!]_\xi\}, true \rangle$

A.3.6. Assignments in SCL Models

Definition A.47 (Syntax of assignments in SCL models). *The syntax of* $\sigma^{imm}_{ass}(\omega_{scl}) \in \Sigma^{imm}_{ass}(\omega_{scl})$ *and* $\sigma^{del}_{ass}(\omega_{scl}) \in \Sigma^{del}_{ass}(\omega_{scl})$ *is defined as follows:*

- $\sigma^{imm}_{ass}(\omega_{scl}) \stackrel{def}{=} \{\ [\texttt{do}]\ x\ =\ \tau[;]\ \}$
- $\sigma^{del}_{ass}(\omega_{scl}) \stackrel{def}{=} \{\ [\texttt{do}]\ x\ =\ \tau(x)[;]\ \}$

Definition A.48 (SOS transition rules of assignments in SCL models). *The SOS transition rules of* $\sigma^{imm}_{ass}(\omega_{scl}) \in \Sigma^{imm}_{ass}(\omega_{scl})$ *and* $\sigma^{del}_{ass}(\omega_{scl}) \in \Sigma^{del}_{ass}(\omega_{scl})$ *are defined as follows:*

- $\langle \xi, \sigma^{imm}_{ass}(\omega_{scl}) \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{x = [\![\tau]\!]_\xi\}, true \rangle$
- $\langle \xi, \sigma^{del}_{ass}(\omega_{scl}) \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{x = [\![\tau(x)]\!]_\xi\}, true \rangle$

A.3.7. Conditions in SCL Models

Definition A.49 (Syntax of conditions in SCL models). *The syntax of* $\sigma^{it}_{cond}(\omega_{scl}) \in \Sigma^{it}_{cond}(\omega_{scl})$ *and* $\sigma^{ite}_{cond}(\omega_{scl}) \in \Sigma^{ite}_{cond}(\omega_{scl})$ *is defined as follows:*

- $\sigma^{it}_{cond}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{if}(\lambda^b(\sigma^{it}_{cond}(\omega_{scl}))) \texttt{\{} \\ \quad \Sigma_1(\sigma^{it}_{cond}(\omega_{scl})) \\ \texttt{\}} \end{array} \right\}$
- $\sigma^{ite}_{cond}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{if}(\lambda^b(\sigma^{ite}_{cond}(\omega_{scl}))) \texttt{\{} \\ \quad \Sigma_1(\sigma^{ite}_{cond}(\omega_{scl})) \\ \texttt{\}else\{} \\ \quad \Sigma_2(\sigma^{ite}_{cond}(\omega_{scl})) \\ \texttt{\}} \end{array} \right\}$

Definition A.50 (SOS transition rules of conditions in SCL models). *The SOS transition rules of* $\sigma^{it}_{cond}(\omega_{scl}) \in \Sigma^{it}_{cond}(\omega_{scl})$ *and* $\sigma^{ite}_{cond}(\omega_{scl}) \in \Sigma^{ite}_{cond}(\omega_{scl})$ *are defined as follows:*

SOS transition rules of $\sigma^{it}_{cond}(\omega_{scl}) \in \Sigma^{it}_{cond}(\omega_{scl})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}{\langle \xi, \{\texttt{if}(\lambda^b)\{\Sigma_1\}\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}{\langle \xi, \{\texttt{if}(\lambda^b)\{\Sigma_1\}\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}$$

SOS transition rules of $\sigma^{ite}_{cond}(\omega_{scl}) \in \Sigma^{ite}_{cond}(\omega_{scl})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{if}(\lambda^b)\{\Sigma_1 \\ \texttt{\}else\{}\Sigma_2\texttt{\}} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{if}(\lambda^b)\{\Sigma_1 \\ \texttt{\}else\{}\Sigma_2\texttt{\}} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}$$

A.3.8. Loops in SCL Models

Definition A.51 (Syntax of loops in SCL models). *The syntax of* $\sigma^{foot}_{loop}(\omega_{scl}) \in \Sigma^{foot}_{loop}(\omega_{scl})$ *and* $\sigma^{head}_{loop}(\omega_{scl}) \in \Sigma^{head}_{loop}(\omega_{scl})$ *is defined as follows:*

- $\sigma^{foot}_{loop}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{do:} \\ \quad \Sigma(\sigma^{foot}_{loop}(\omega_{scl}) \\ \quad \texttt{pause;} \\ \texttt{if}(\lambda^b(\sigma^{foot}_{loop}(\omega_{scl}))) \texttt{\{} \\ \quad \texttt{goto do;} \\ \texttt{\}} \end{array} \right\}$

- $\sigma^{head}_{loop}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{while}(\lambda^b(\sigma^{head}_{loop}(\omega_{scl}))\{ \\ \quad \Sigma(\sigma^{head}_{loop}(\omega_{scl}) \\ \quad \texttt{pause;} \\ \} \end{array} \right\}$
- $\sigma^{inf}_{loop}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{loop:} \\ \quad \Sigma(\sigma^{inf}_{loop}(\omega_{scl})) \\ \quad \texttt{goto loop;} \end{array} \right\}$
  - $\sigma^{inf}_{loop}(\omega_{scl}) :\equiv \sigma^{foot}_{loop}(\omega_{scl}) \iff \lambda^b(\sigma^{foot}_{loop}(\omega_{scl})) = true$

Definition A.52 (SOS transition rules of loops in SCL models). *The SOS transition rules of $\sigma^{foot}_{loop}(\omega_{scl}) \in \Sigma^{foot}_{loop}(\omega_{scl})$ and $\sigma^{head}_{loop}(\omega_{scl}) \in \Sigma^{head}_{loop}(\omega_{scl})$ are defined as follows:*

SOS transition rules of $\sigma^{foot}_{loop}(\omega_{scl}) \in \Sigma^{foot}_{loop}(\omega_{scl})$:

- $$\frac{\langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle \Sigma', \mathcal{D}, false \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{do:}\ \Sigma \\ \texttt{if}(\lambda^b)\ \texttt{goto do;} \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \{\Sigma';\ \texttt{while}(\lambda^b)\ \Sigma\}, \mathcal{D}, false \rangle}$$

SOS transition rules of $\sigma^{head}_{loop}(\omega_{scl}) \in \Sigma^{head}_{loop}(\omega_{scl})$:

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle \Sigma', \mathcal{D}, false \rangle}{\langle \xi, \{\texttt{while}(\lambda^b)\ \Sigma\} \rangle \xrightarrow{\mathbb{T}} \langle \{\Sigma';\ \texttt{while}(\lambda^b)\ \Sigma\}, \mathcal{D}, false \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false}{\langle \xi, \{\texttt{while}(\lambda^b)\ \Sigma\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, true \rangle}$$

A.3.9. Pause Statements in SCL Models

Definition A.53 (Syntax of pause statements in SCL models). *The syntax of pause statements $\Sigma_{pause}(\omega_{scl})$ is defined as follows:*

- $\Sigma_{pause}(\omega_{scl}) \stackrel{def}{=} \texttt{pause;}$

Definition A.54 (SOS transition rules of pause statements in SCL models). *The SOS transition rules of pause statements $\Sigma_{pause}(\omega_{scl})$ are defined as follows:*

- $\langle \xi, \texttt{pause;} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \{\}, false \rangle$

A.3.10. Sequences in SCL Models

Definition A.55 (Syntax of sequences in SCL models). *The syntax of a sequence $\Sigma_{seq}(\omega_{scl})$ is defined as follows:*

- $\Sigma_{seq}(\omega_{scl}) \stackrel{def}{=} \left\{ \begin{array}{l} \sigma_1(\omega_{scl})\,; \\ \sigma_2(\omega_{scl})\,; \\ \quad\vdots \\ \sigma_n(\omega_{scl})\,; \end{array} \right\}$

Definition A.56 (SOS transition rules of sequences in SCL models). *The SOS transition rules of a sequence* $\Sigma_{seq}(\omega_{scl})$ *are defined as follows:*

- $$\frac{\langle \xi, \sigma_1 \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \sigma_1', \mathcal{D}_1, false \rangle}{\langle \xi, \{\sigma_1; \sigma_2;\} \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \{\sigma_1'; \sigma_2;\}, \mathcal{D}_1, false \rangle}$$
- $$\frac{\langle \xi, \sigma_1 \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \sigma_1', \mathcal{D}_1, true \rangle \wedge \langle \xi', \sigma_2 \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \{\sigma_1; \sigma_2;\} \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \sigma_2', \{\mathcal{D}_1; \mathcal{D}_2\}, f_2 \rangle}$$

A.4. Data-Flow Oriented SCCharts

A.4.1. Data-Flow Oriented SCCharts Declaration

Definition A.57 (Syntax of data-flow oriented SCChart elements). *A data-flow oriented SCChart is declared as follows, assuming fixed order of* $\Delta_{idcl}(\omega_{scd})$, $\Delta_{vdcl}(\omega_{scd})$, *and* $\Sigma(\omega_{scd})$*:*

- $\delta_\omega(\omega_{scd}) \stackrel{def}{=} \left\{ \begin{array}{l} [\Delta_{imports}(\omega_{scd})] \\ \texttt{scchart}\ a_n(\omega_{scd})\texttt{\{} \\ \quad [\Delta_{idcl}(\omega_{scd})] \\ \quad [\Delta_{vdcl}(\omega_{scd})] \\ \quad \texttt{dataflow:} \\ \quad\quad [\Sigma(\omega_{scd})] \\ \texttt{\}} \end{array} \right\}$

Definition A.58 (Semantics of data-flow oriented SCChart elements). *The semantics of* ω_{scd} *are defined as follows:*

- $[\![\delta_\omega(\omega_{scd})]\!]_\xi \stackrel{def}{=} \{$ *defines a data-flow oriented SCChart* ω_{scd} *with* $\Sigma(\omega_{scd})$*, started at time* t *with initial conditions set by* $\Delta_{idcl}(\omega_{scd})$ *and* $\Delta_{vdcl}(\omega_{scd})$ *when invoked, and preserving internal state across* macro steps*. Furthermore,* ω_{scd} *never terminates.* $\}$

A.4.2. Local Variables in SCCharts

Definition A.59 (Syntax of local variables in SCCharts). *The syntax of* $\Delta_{local}(\omega_{scd})$ *and* $\Delta_{inst}(\omega_{scd})$ *is defined as follows:*

- $\Delta_{local}(\omega_{scd}) \stackrel{def}{=} \left\{ \begin{array}{l} \alpha_1^{[+]}\ \ x_1\texttt{[=}w_1\texttt{][;]} \\ \vdots \\ \alpha_n^{[+]}\ \ x_n\texttt{[=}w_n\texttt{][;]} \end{array} \right\}$

- $\Delta_{inst}(\omega_{scd}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{ref}\ \ x_1\ \ k_1 \\ \vdots \\ \texttt{ref}\ \ x_n\ \ k_n \end{array} \right\}$

Definition A.60 (Semantics of local variables in SCCharts). *The semantics of $\Delta_{local}(\omega_{scd})$ and $\Delta_{inst}(\omega_{scd})$ are defined as follows:*

- $[\![\Delta_{local}(\omega_{scd})]\!]_\xi \stackrel{def}{=} \{$ *defines a set of variables $x_1, \ldots, x_n$ with corresponding SCCharts $[\![\alpha]\!]_\xi, \ldots, [\![\alpha_n^{[+]}]\!]_\xi$. These variables are assigned to optional predefined default values $[\![w_1]\!]_\xi, \ldots, [\![w_n]\!]_\xi$ at time t when ω_{scd} is invoked and keep their values when switching from one* macro step *to another. These variables can be modified and processed locally until ω_{scd} terminates at time $t + \theta$, i.e., all instructions of ω_{scd} have been processed.* $\}$
- $[\![\Delta_{inst}(\omega_{scd})]\!]_\xi \stackrel{def}{=} \{$ *defines a set of variables $k_1, \ldots, k_n$ of SCCharts $[\![x_1]\!]_\xi, \ldots, [\![x_n]\!]_\xi$. These instances are invoked at time t when ω_{scd} is invoked and keep their values when switching from one* macro step *to another.* $\}$

A.4.3. SCChart Imports

Definition A.61 (Syntax of SCChart imports). *Imported SCCharts are grouped as a set $\Delta_{imports}(\omega_{scd})$, whose syntax is defined as follows:*

- $\Delta_{imports}(\omega_{scd}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{import "}x_1\texttt{.sctx";} \\ \vdots \\ \texttt{import "}x_n\texttt{.sctx";} \end{array} \right\}$

Definition A.62 (Semantics of SCChart imports). *The semantics of $\Delta_{imports}(\omega_{scd})$ are defined as follows:*

- $[\![\Delta_{imports}(\omega_{scd})]\!]_\xi \stackrel{def}{=} \{$ *defines a set of imported SCCharts $x_1, \ldots, x_n$ that can be instantiated and invoked by ω_{scd}.* $\}$

A.4.4. Synchronous Concurrency in Data-Flow Oriented SCCharts

Definition A.63 (Syntax of synchronous concurrency in data-flow oriented SCCharts). *The syntax of a synchronous parallel statement $\Sigma_{conc}(\omega_{scd})$ is defined as follows:*

- $\Sigma_{conc}(\omega_{scd}) \stackrel{def}{=} \Sigma_1(\omega_{scd})\ \ ;\ \ \Sigma_2(\omega_{scd})$

Definition A.64 (SOS transition rules of synchronous concurrency in data-flow oriented SCCharts). *The SOS transition rules of a synchronous parallel statement $\Sigma_{conc}(\omega_{scd})$ are defined as follows:*

- $$\frac{\langle \xi, \Sigma_1 \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle \wedge \langle \xi, \Sigma_2 \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \{\Sigma_1\ \ \|\ \ \Sigma_2\} \rangle \stackrel{\mathbb{T}}{\longrightarrow\!\!\!\!\!\rightarrow} \langle \{\Sigma_1'\ \ \|\ \ \Sigma_2'\}, \{\mathcal{D}_1; \mathcal{D}_2\}, f_1 \wedge f_2 \rangle}$$

A.4.5. Module Invocations in Data-Flow Oriented SCCharts

Definition A.65 (Syntax of SCChart invocations). *The syntax of an SCChart invocation* $\sigma^{fb}_{inv}(\omega_{qrz}) \in \Sigma^{fb}_{inv}(\omega_{qrz})$ *is defined as follows, where* k *the instance name:*

- $\sigma^{fb}_{inv}(\omega_{qrz}) \stackrel{def}{=} \left\{ k\texttt{=\{}\mathcal{I}(\sigma^{fb}_{inv}(\omega_{qrz}))\texttt{\};} \right\}$ *denotes a* Quartz *module invocations of an instantiated module*

Definition A.66 (SOS transition rules of SCChart invocations). *The SOS transition rules of an SCChart invocation* $\sigma^{fb}_{inv}(\omega_{qrz}) \in \Sigma^{fb}_{inv}(\omega_{qrz})$ *are defined as follows:*

SOS transition rules of $\sigma^{fb}_{inv}(\omega_{qrz}) \in \Sigma^{fb}_{inv}(\omega_{qrz})$*:*

- $\langle \xi, \sigma^{fb}_{inv} \rangle \xrightarrow{\mathbb{T}} \langle \sigma^{fb'}_{inv}, [\![k]\!]_\xi, f \rangle$

A.4.6. Sequences in Data-Flow Oriented SCCharts

Definition A.67 (Syntax of sequences in Data-Flow Oriented SCCharts). *The syntax of a sequence* $\Sigma_{seq}(\omega_{scd})$ *is defined as follows:*

- $\Sigma_{seq}(\omega_{scd}) \stackrel{def}{=} \left\{ \begin{array}{l} \sigma_1(\omega_{qrz})\texttt{;} \\ \sigma_2(\omega_{qrz})\texttt{;} \\ \quad\vdots \\ \sigma_n(\omega_{qrz})\texttt{;} \end{array} \right\}$

Definition A.68 (SOS transition rules of sequences in Data-Flow Oriented SCCharts). *The SOS transition rules of a sequence* $\Sigma_{seq}(\omega_{scd})$ *are defined as follows:*

- $$\frac{\langle \xi, \sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \sigma_1', \mathcal{D}_1, f_1 \rangle \wedge \langle \xi, \sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \{\sigma_1; \sigma_2\} \rangle \xrightarrow{\mathbb{T}} \langle \{\sigma_1'; \sigma_2'\}, \{\mathcal{D}_1; \mathcal{D}_2\}, f_1 \wedge f_2 \rangle}$$

A.5. Control-Flow Oriented SCCharts

A.5.1. Control-Flow Oriented SCCharts Declaration

Definition A.69 (Syntax of control-flow oriented SCChart elements). *A control-flow oriented SCChart is declared as follows, assuming fixed order of* $\Delta_{idcl}(\omega_{scc})$*,* $\Delta_{vdcl}(\omega_{scc})$*, and* $\Sigma(\omega_{scc})$*:*

- $\delta_\omega(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{scchart } a_n(\omega_{scc})\texttt{\{} \\ \quad [\Delta_{idcl}(\omega_{scc})] \\ \quad [\Delta_{vdcl}(\omega_{scc})] \\ \quad \texttt{region:} \\ \quad\quad [\Sigma(\omega_{scc})] \\ \quad [\texttt{region: ...}] \\ \texttt{\}} \end{array} \right\}$

Definition A.70 (Semantics of control-flow oriented SCChart elements). *The semantics of ω_{scc} are defined as follows:*

- $[\![\delta_\omega(\omega_{scc})]\!]_\xi \stackrel{def}{=} \{$ *defines a control-flow oriented SCChart ω_{scc} with $\Sigma(\omega_{scc})$, started at time t with initial conditions set by $\Delta_{idcl}(\omega_{scc})$ and $\Delta_{vdcl}(\omega_{scc})$ when invoked, and preserving internal state across* macro steps.

A.5.2. Abortions in control-flow oriented SCCharts

Definition A.71 (Syntax of abortions in control-flow oriented SCCharts). *The syntax of a strong delayed abortion $\Sigma^{reg}_{abort}(\omega_{scc})$ and a strong immediate abortion $\Sigma^{imm}_{abort}(\omega_{scc})$ is defined as follows:*

- $\sigma^{reg}_{abort}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{reg}_{abort}(\omega_{scc})) \\ \texttt{if } \lambda^b(\sigma^{reg}_{abort}(\omega_{scc})) \texttt{ abort to } \Sigma_2(\sigma^{reg}_{abort}(\omega_{scc})) \\ \texttt{[final] state } \Sigma_2(\sigma^{reg}_{abort}(\omega_{scc})) \end{array} \right\}$ *denotes a strong delayed abortion*
- $\sigma^{imm}_{abort}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{imm}_{abort}(\omega_{scc})) \\ \texttt{immediate if } \lambda^b(\sigma^{imm}_{abort}(\omega_{scc})) \texttt{ abort to} \\ \quad \Sigma_2(\sigma^{imm}_{abort}(\omega_{scc})) \\ \texttt{[final] state } \Sigma_2(\sigma^{imm}_{abort}(\omega_{scc})) \end{array} \right\}$ *denotes a strong immediate abortion*

Definition A.72 (SOS transition rules of abortions in control-flow oriented SCCharts). *The SOS transition rules of a strong delayed abortion $\Sigma^{reg}_{abort}(\omega_{scc})$ and a strong immediate abortion $\Sigma^{imm}_{abort}(\omega_{scc})$ are defined as follows:*

SOS transition rules of $\sigma^{reg}_{abort}(\omega_{scc}) \in \Sigma^{reg}_{abort}(\omega_{scc})$:

- $$\frac{\langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{if } \lambda^b \texttt{ abort } \textit{to} \ \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}$$
- $$\frac{\langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, false \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{if } \lambda^b \texttt{ abort } \textit{to} \ \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1' \\ \texttt{immediate if } \lambda^b \texttt{ abort } \textit{to} \ \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\}, \mathcal{D}_1, false \rangle}$$

SOS transition rules of $\sigma^{imm}_{abort}(\omega_{scc}) \in \Sigma^{imm}_{abort}(\omega_{scc})$:

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \Sigma_2, \mathcal{D}_2, f_2 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{immediate if } \lambda^b \texttt{ abort } \textit{to} \ \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{immediate if } \lambda^b \texttt{ abort } \textit{to} \ \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle nothing, \mathcal{D}_1, true \rangle}$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, false \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{immediate if } \lambda^b \texttt{ abort } \textit{to} \ \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1' \\ \texttt{immediate if } \lambda^b \texttt{ abort } \textit{to} \ \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\}, \mathcal{D}_1, false \rangle}$$

A.5.3. Await Transitions in control-flow oriented SCCharts

Definition A.73 (Syntax and SOS transition rules of await transitions in control-flow oriented SCCharts). *The syntax of a delayed await transition* $\Sigma^{reg}_{await}(\omega_{scc})$ *and a immediate await transition* $\Sigma^{imm}_{await}(\omega_{scc})$ *is defined as follows. The SOS transitions rules are derived from equivalent constructs:*

- $\sigma^{reg}_{await}(\omega_{scc}) :\equiv \sigma^{reg}_{abort}(\omega_{scc})$
- $\sigma^{imm}_{await}(\omega_{scc}) :\equiv \sigma^{imm}_{abort}(\omega_{scc})$

A.5.4. Synchronous Concurrency in control-flow oriented SCCharts

Definition A.74 (Syntax of synchronous concurrency in control-flow oriented SCCharts). *The syntax of a synchronous parallel statement* $\Sigma_{conc}(\omega_{scc})$ *is defined as follows:*

- $$\Sigma_{conc}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{region:} \\ \quad [\Sigma_1(\omega_{scc})] \\ \texttt{region:} \\ \quad [\Sigma_2(\omega_{scc})] \end{array} \right\}$$

Definition A.75 (SOS transition rules of synchronous concurrency in control-flow oriented SCCharts). *The SOS transition rules of a synchronous parallel statement* $\Sigma_{conc}(\omega_{scc})$ *are defined as follows:*

- $$\frac{\langle \xi, \Sigma_1 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_1', \mathcal{D}_1, f_1 \rangle \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \{\Sigma_1 \parallel \Sigma_2\} \rangle \xrightarrow{\mathbb{T}} \langle \{\Sigma_1' \parallel \Sigma_2'\}, \{\mathcal{D}_1 \cup \mathcal{D}_2\}, f_1 \wedge f_2 \rangle}$$

A.5.5. Conditions in control-flow oriented SCCharts

Definition A.76 (Syntax of conditions in control-flow oriented SCCharts). *The syntax of* $\sigma^{it}_{cond}(\omega_{scc}) \in \Sigma^{it}_{cond}(\omega_{scc})$ *and* $\sigma^{ite}_{cond}(\omega_{scc}) \in \Sigma^{ite}_{cond}(\omega_{scc})$ *is defined as follows:*

- $\sigma^{it}_{cond}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{it}_{cond}(\omega_{scc})) \\ \texttt{immediate if } \lambda^b(\sigma^{it}_{cond}(\omega_{scc})) \texttt{ go} \\ \quad \texttt{to } \Sigma_2(\sigma^{it}_{cond}(\omega_{scc})) \\ \texttt{immediate if !}(\lambda^b(\sigma^{it}_{cond}(\omega_{scc}))) \texttt{ go} \\ \quad \texttt{to } \Sigma_3(\sigma^{it}_{cond}(\omega_{scc})) \\ \texttt{state } \Sigma_2(\sigma^{it}_{cond}(\omega_{scc})) \\ \texttt{[...] } \Sigma_3(\sigma^{it}_{cond}(\omega_{scc})) \\ \texttt{[final] state } \Sigma_3(\sigma^{it}_{cond}(\omega_{scc})) \end{array} \right\}$
- $\sigma^{ite}_{cond}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{ite}_{cond}(\omega_{scc})) \\ \texttt{immediate if } \lambda^b(\sigma^{ite}_{cond}(\omega_{scc})) \texttt{ go} \\ \quad \texttt{to } \Sigma_2(\sigma^{ite}_{cond}(\omega_{scc})) \\ \texttt{immediate if !}(\lambda^b(\sigma^{ite}_{cond}(\omega_{scc}))) \texttt{ go} \\ \quad \texttt{to } \Sigma_3(\sigma^{ite}_{cond}(\omega_{scc})) \\ \texttt{state } \Sigma_2(\sigma^{ite}_{cond}(\omega_{scc})) \\ \texttt{[...] } \Sigma_4(\sigma^{ite}_{cond}(\omega_{scc})) \\ \texttt{state } \Sigma_3(\sigma^{ite}_{cond}(\omega_{scc})) \\ \texttt{[...] } \Sigma_4(\sigma^{ite}_{cond}(\omega_{scc})) \\ \texttt{[final] state } \Sigma_4(\sigma^{ite}_{cond}(\omega_{scc})) \end{array} \right\}$

Definition A.77 (SOS transition rules of conditions in control-flow oriented SCCharts). *The SOS transition rules of* $\sigma^{it}_{cond}(\omega_{scc}) \in \Sigma^{it}_{cond}(\omega_{scc})$ *and* $\sigma^{ite}_{cond}(\omega_{scc}) \in \Sigma^{ite}_{cond}(\omega_{scc})$ *are defined as follows:*

SOS transition rules of $\sigma^{it}_{cond}(\omega_{scc}) \in \Sigma^{it}_{cond}(\omega_{scc})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{immediate if } \lambda^b \texttt{ go} \\ \quad \texttt{to } \Sigma_2 \\ \texttt{immediate if !}(\lambda^b) \texttt{ go} \\ \quad \texttt{to } \Sigma_3 \\ \texttt{state } \Sigma_2 \\ \texttt{[...] } \Sigma_3 \\ \texttt{[final] state } \Sigma_3 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \{\Sigma_2'; \Sigma_3\}, \mathcal{D}_2, f_2 \rangle}$$
- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_3 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_3', \mathcal{D}_3, f_3 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{immediate if } \lambda^b \texttt{ go} \\ \quad \texttt{to } \Sigma_2 \\ \texttt{immediate if !}(\lambda^b) \texttt{ go} \\ \quad \texttt{to } \Sigma_3 \\ \texttt{state } \Sigma_2 \\ \texttt{[...] } \Sigma_3 \\ \texttt{[final] state } \Sigma_3 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_3', \mathcal{D}_3, f_3 \rangle}$$

SOS transition rules of $\sigma^{ite}_{cond}(\omega_{scc}) \in \Sigma^{ite}_{cond}(\omega_{scc})$*:*

- $$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1) \\ \texttt{immediate if } \lambda^b \texttt{ go} \\ \quad \texttt{to } \Sigma_2 \\ \texttt{immediate if } !(\lambda^b) \texttt{ go} \\ \quad \texttt{to } \Sigma_3 \\ \texttt{state } \Sigma_2 \\ [...]\ \Sigma_4 \\ \texttt{state } \Sigma_3 \\ [...]\ \Sigma_4 \\ \texttt{[final] state } \Sigma_4 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \{\Sigma_2'; \Sigma_4\}, \mathcal{D}_2, f_2 \rangle}$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_3 \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \Sigma_3', \mathcal{D}_3, f_3 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{immediate if } \lambda^b \texttt{ go} \\ \quad \texttt{to } \Sigma_2 \\ \texttt{immediate if } !(\lambda^b) \texttt{ go} \\ \quad \texttt{to } \Sigma_3 \\ \texttt{state } \Sigma_2 \\ [...]\ \Sigma_4 \\ \texttt{state } \Sigma_3 \\ [...]\ \Sigma_4 \\ \texttt{[final] state } \Sigma_4 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \{\Sigma_3'; \Sigma_4\}, \mathcal{D}_3, f_3 \rangle}$$

A.5.6. Halt Statements in control-flow oriented SCCharts

Definition A.78 (Syntax and SOS transitions rules of halt statements in control-flow oriented SCCharts). *The syntax of halt statements* $\Sigma_{halt}(\omega_{scc})$ *is defined as follows. The SOS transitions rules are derived from equivalent constructs [Sch09]:*

- $\sigma_{halt}(\omega_{scc}) \stackrel{def}{=} \iff \lambda^b(\sigma_{loop}^{foot}(\omega_{scc})) = true \wedge \Sigma_1(\sigma_{loop}^{foot}(\omega_{scc})) = \Sigma_{pause}(\omega_{scc}))$

A.5.7. Loops in control-flow oriented SCCharts

Definition A.79 (Syntax of loops in control-flow oriented SCCharts). *The syntax of* $\sigma_{loop}^{foot}(\omega_{scc}) \in \Sigma_{loop}^{foot}(\omega_{scc})$, $\sigma_{loop}^{head}(\omega_{scc}) \in \Sigma_{loop}^{head}(\omega_{scc})$, *and* $\sigma_{loop}^{inf}(\omega_{scc}) \in \Sigma_{loop}^{inf}(\omega_{scc})$ *is defined as follows:*

- $\sigma^{foot}_{loop}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma^{foot}_{loop}(\omega_{scc})) \\ \ldots \\ \texttt{state } \Sigma_n(\sigma^{foot}_{loop}(\omega_{scc})) \\ \texttt{immediate if } \lambda^b(\sigma^{foot}_{loop}(\omega_{scc})) \\ \quad \texttt{go to } \Sigma_1(\sigma^{foot}_{loop}(\omega_{scc})) \\ \texttt{immediate if } !(\lambda^b(\sigma^{foot}_{loop}(\omega_{scc})) \\ \quad \texttt{join to } \Sigma_2(\sigma^{foot}_{loop}(\omega_{scc})) \\ \texttt{[final] state } \Sigma_2(\sigma^{foot}_{loop}(\omega_{scc})) \end{array} \right\}$

- $\sigma^{head}_{loop}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_i(\sigma^{head}_{loop}(\omega_{scc})) \\ \texttt{immediate if } \lambda^b(\sigma^{head}_{loop}(\omega_{scc})) \\ \quad \texttt{go to } \Sigma_1(\sigma^{head}_{loop}(\omega_{scc})) \\ \texttt{immediate if } !(\lambda^b(\sigma^{head}_{loop}(\omega_{scc})) \\ \quad \texttt{join to } \Sigma_2(\sigma^{head}_{loop}(\omega_{scc})) \\ \texttt{state } \Sigma_1(\sigma^{head}_{loop}(\omega_{scc})) \\ \ldots \\ \texttt{state } \Sigma_n(\sigma^{head}_{loop}(\omega_{scc})) \\ \texttt{go to } \Sigma_i(\sigma^{head}_{loop}(\omega_{scc})) \\ \texttt{[final] state } \Sigma_2(\sigma^{head}_{loop}(\omega_{scc})) \end{array} \right\}$

- $\sigma^{inf}_{loop}(\omega_{scc}) \stackrel{def}{=} \sigma^{foot}_{loop}(\omega_{scc}) \iff \lambda^b(\sigma^{foot}_{loop}(\omega_{scc})) = true$

Definition A.80 (SOS transition rules of loops in control-flow oriented SCCharts). *The SOS transition rules of* $\sigma^{foot}_{loop}(\omega_{scc}) \in \Sigma^{foot}_{loop}(\omega_{scc})$ *and* $\sigma^{head}_{loop}(\omega_{scc}) \in \Sigma^{head}_{loop}(\omega_{scc})$ *are defined as follows:*

SOS transition rules of $\sigma^{foot}_{loop}(\omega_{scc}) \in \Sigma^{foot}_{loop}(\omega_{scc})$*:*

$$\frac{\langle \xi, \Sigma_n \rangle \xrightarrow{\mathbb{T}}\!\!\!\!\rightarrow \langle \Sigma'_n, \mathcal{D}_n, f_n \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \ldots \\ \texttt{state } \Sigma_n \\ \texttt{immediate if } \lambda^b \\ \quad \texttt{go to } \Sigma_1 \\ \texttt{immediate if } !(\lambda^b) \\ \quad \texttt{join to } \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}}\!\!\!\!\rightarrow}$$

$$\bullet \quad \langle \{\Sigma'_n; \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_i \\ \texttt{immediate if } \lambda^b \\ \quad \texttt{go to } \Sigma_1 \\ \texttt{immediate if } !(\lambda^b) \\ \quad \texttt{join to } \Sigma_2 \\ \texttt{state } \Sigma_1 \\ \ldots \\ \texttt{state } \Sigma_n \\ \texttt{go to } \Sigma_i \\ \texttt{[final] state } \Sigma_2 \end{array} \right\}, \mathcal{D}_n, f_n \rangle$$

SOS transition rules of $\sigma^{head}_{loop}(\omega_{scc}) \in \Sigma^{head}_{loop}(\omega_{scc})$*:*

$$\frac{[\![\lambda^b]\!]_\xi = true \wedge \langle \xi, \Sigma_i \rangle \xrightarrow{\mathbb{T}}\!\!\!\!\rightarrow \langle \Sigma'_i, \mathcal{D}_i, f_i \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_i \\ \texttt{immediate if } \lambda^b \\ \quad \texttt{go to } \Sigma_1 \\ \texttt{immediate if } !(\lambda^b) \\ \quad \texttt{join to } \Sigma_2 \\ \texttt{state } \Sigma_1 \\ \ldots \\ \texttt{state } \Sigma_n \\ \texttt{go to } \Sigma_i \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}}\!\!\!\!\rightarrow}$$

$$\bullet \quad \langle \{\Sigma'_i; \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_i \\ \texttt{immediate if } \lambda^b \\ \quad \texttt{go to } \Sigma_1 \\ \texttt{immediate if } !(\lambda^b) \\ \quad \texttt{join to } \Sigma_2 \\ \texttt{state } \Sigma_1 \\ \ldots \\ \texttt{state } \Sigma_n \\ \texttt{go to } \Sigma_i \\ \texttt{[final] state } \Sigma_2 \end{array} \right\}, \mathcal{D}_i, f_i \rangle$$

- $$\frac{[\![\lambda^b]\!]_\xi = false \wedge \langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_i \\ \texttt{immediate if } \lambda^b \\ \quad \texttt{go to } \Sigma_1 \\ \texttt{immediate if } !(\lambda^b) \\ \quad \texttt{join to } \Sigma_2 \\ \texttt{state } \Sigma_1 \\ \ldots \\ \texttt{state } \Sigma_n \\ \texttt{go to } \Sigma_i \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}$$

A.5.8. Immediate Transitions in control-flow oriented SCCharts

Definition A.81 (Syntax of immediate transitions in control-flow oriented SCCharts). *The syntax of an immediate transition $\Sigma_{nothing}(\omega_{scc})$ is defined as follows:*

- $$\sigma_{nothing}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1(\sigma_{nothing}(\omega_{scc})) \\ \texttt{immediate abort to } \Sigma_2(\sigma_{nothing}(\omega_{scc})) \\ \texttt{[final] state } \Sigma_2(\sigma_{nothing}(\omega_{scc})) \end{array} \right\}$$

Definition A.82 (SOS transition rules of immediate transitions in control-flow oriented SCCharts). *The SOS transition rules of an immediate transition $\Sigma_{nothing}(\omega_{scc})$ are defined as follows:*

- $$\frac{\langle \xi, \Sigma_2 \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \left\{ \begin{array}{l} \texttt{[initial] state } \Sigma_1 \\ \texttt{immediate abort to } \Sigma_2 \\ \texttt{[final] state } \Sigma_2 \end{array} \right\} \rangle \xrightarrow{\mathbb{T}} \langle \Sigma_2', \mathcal{D}_2, f_2 \rangle}$$

A.5.9. Pause Statements in control-flow oriented SCCharts

Definition A.83 (Syntax and SOS transition rules of pause statements in control-flow oriented SCCharts). *The syntax of pause statements $\Sigma_{pause}(\omega_{scc})$ is defined as follows. The SOS transitions rules are derived from equivalent constructs:*

- $\sigma_{pause}(\omega_{scc}) :\equiv \sigma^{reg}_{await}(\omega_{scc}) \iff \lambda^b(\sigma^{reg}_{await}(\omega_{scc})) = true$

A.5.10. Sequences in control-flow oriented SCCharts

Definition A.84 (Syntax of sequences in control-flow oriented SCCharts). *The syntax of a sequence $\Sigma_{seq}(\omega_{scc})$ is defined as follows:*

- $$\Sigma_{seq}(\omega_{scc}) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{initial state } \sigma_i(\sigma_{seq}(\omega_{scc})) \\ \texttt{immediate [go|join] to } \sigma_j(\sigma_{seq}(\omega_{scc})) \\ \texttt{[final] state } \sigma_j(\sigma_{seq}(\omega_{scc})) \end{array} \right\}$$

Definition A.85 (SOS transition rules of sequences in control-flow oriented SCCharts). *The SOS transition rules of a sequence* $\Sigma_{seq}(\omega_{scc})$ *are defined as follows:*

- $$\frac{\langle \xi, \sigma_1 \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \sigma_1', \mathcal{D}_1, f_1 \rangle \wedge \langle \xi, \sigma_2 \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \sigma_2', \mathcal{D}_2, f_2 \rangle}{\langle \xi, \{\sigma_1; \sigma_2\} \rangle \xrightarrow{\mathbb{T}} \!\!\!\!\rightarrow \langle \{\sigma_1'; \sigma_2'\}, \{\mathcal{D}_1; \mathcal{D}_2\}, f_1 \wedge f_2 \rangle}$$

Appendix B

ST Model Examples

```
PROGRAM ST_TWO_OF_THREE
  VAR_INPUT
    xB1_Temp : BOOL;
    xB2_Temp : BOOL;
    xB3_Temp : BOOL;
  END_VAR
  VAR_OUTPUT
    xP1_Temp : BOOL;
  END_VAR

  xP1_Temp := xB1_Temp AND xB2_Temp OR xB1_Temp AND xB3_Temp OR xB2_Temp
      AND xB3_Temp;
END_PROGRAM
```

Listing B.1: *ST Model: ST_TWO_OF_THREE*

```
FUNCTION ST_ALARM : BOOL
  VAR_INPUT
    xSENSOR_L : BOOL;
    xSENSOR_M : BOOL;
    xSENSOR_R : BOOL;
  END_VAR

  ST_ALARM := (NOT xSENSOR_L AND NOT xSENSOR_M AND NOT xSENSOR_R) OR (
      xSENSOR_L AND xSENSOR_R);
END_FUNCTION
```

Listing B.2: *ST Model: ST_ALARM*

```
FUNCTION ST_SCALE : REAL
  VAR_INPUT
    iX : INT;
    rY_MAX : REAL;
    rY_MIN : REAL;
  END_VAR

  ST_SCALE := (rY_MAX - rY_MIN) / 32760.0 * iX + rY_MIN;
END_FUNCTION
```

Listing B.3: *ST Model: ST_SCALE*

```
FUNCTION_BLOCK ST_AVAL_PROC
  VAR
    rPressure : REAL;
```

```
    iPressure_Per : INT;
  END_VAR

  ST_SCALE(iX := iPressure_Per, rY_MAX := 10.0, rY_MIN := 6.0, ST_SCALE
      => rPressure);
END_FUNCTION_BLOCK
```

Listing B.4: *ST Model:* `ST_AVAL_PROC`

```
FUNCTION_BLOCK ST_OP_ARITH
  VAR
    x01 : REAL;
    x02 : REAL;
    x03 : REAL;
    x04 : REAL;
    x05 : REAL;
    x06 : REAL;
    x1 : REAL := 1.0;
    x2 : REAL := 2.0;
    x3 : INT := 1;
    x4 : INT := 2;
  END_VAR

  x01 := x1 + x2;
  x02 := x1 - x2;
  x03 := x1 * x2;
  x04 := x1 / x2;
  x05 := EXPT(x1, x2);
  x06 := x3 MOD x4;
END_FUNCTION_BLOCK
```

Listing B.5: *ST Model:* `ST_OP_ARITH`

```
FUNCTION_BLOCK ST_OP_BOOL
  VAR
    x01 : BOOL;
    x02 : BOOL;
    x03 : BOOL;
    x04 : BOOL;
    x1 : BOOL := TRUE;
    x2 : BOOL := FALSE;
  END_VAR

  x01 := NOT x1;
  x02 := x1 AND x2;
  x03 := x1 OR x2;
  x04 := x1 XOR x2;
END_FUNCTION_BLOCK
```

Listing B.6: *ST Model:* `ST_OP_BOOL`

```
PROGRAM ST_COMPENS
  VAR
    rQ : REAL := 1500.0;
    xQ1 : BOOL;
    xQ2 : BOOL;
    xQ3 : BOOL;
  END_VAR

  IF rQ < 1000.0 THEN
    xQ3 := FALSE;
    xQ2 := FALSE;
    xQ1 := FALSE;
  END_IF;
  IF rQ >= 1000.0 THEN
```

```
    xQ3 := FALSE;
    xQ2 := FALSE;
    xQ1 := TRUE;
  END_IF;
  IF rQ >= 2000.0 THEN
    xQ3 := FALSE;
    xQ2 := TRUE;
    xQ1 := TRUE;
  END_IF;
  IF rQ > 3000.0 THEN
    xQ3 := TRUE;
    xQ2 := TRUE;
    xQ1 := TRUE;
  END_IF;
END_PROGRAM
```

Listing B.7: *ST Model:* `ST_COMPENS`

```
FUNCTION_BLOCK ST_COND
  VAR
    x0 : BOOL;
    x1 : BOOL := TRUE;
    x2 : BOOL := TRUE;
  END_VAR

  IF x1 THEN
    x2 := TRUE;
  END_IF;
  IF x1 THEN
    x0 := TRUE;
  ELSE
    x0 := FALSE;
  END_IF;
END_FUNCTION_BLOCK
```

Listing B.8: *ST Model:* `ST_COND`

```
FUNCTION_BLOCK ST_DATATYPES
  VAR
    A1 : BOOL;
    A2 : BOOL := TRUE;
    A3 : BYTE;
    A4 : WORD;
    A5 : INT;
    A6 : INT := 2;
    A7 : DINT;
    A8 : DINT := 2;
    A9 : UINT;
    A10 : UINT := 2;
    A11 : UDINT;
    A12 : UDINT := 2;
    A13 : REAL;
    A14 : REAL := 1.23;
    A15 : TIME;
    A16 : TIME := T#5000MS;
    A17 : ARRAY [1..2] OF BOOL;
    A18 : ARRAY [1..2] OF BYTE;
    A19 : ARRAY [1..2] OF WORD;
    A20 : ARRAY [1..2] OF INT;
    A21 : ARRAY [1..2] OF DINT;
    A22 : ARRAY [1..2] OF UINT;
    A23 : ARRAY [1..2] OF UDINT;
    A24 : ARRAY [1..2] OF REAL;
    A25 : ARRAY [1..2] OF TIME;
  END_VAR
```

```
END_FUNCTION_BLOCK
```

Listing B.9: *ST Model: ST_DATATYPES*

```
FUNCTION_BLOCK ST_DEBOUNCE
  VAR_INPUT
    IN : BOOL;
    DB_TIME : TIME;
  END_VAR
  VAR_OUTPUT
    OUT : BOOL;
    ET_OFF : TIME;
  END_VAR
  VAR
    DB_ON : TON;
    DB_OFF : TON;
    DB_FF : SR;
  END_VAR

  DB_ON(IN := IN, PT := DB_TIME);
  DB_OFF(IN := NOT IN, PT := DB_TIME);
  ET_OFF := DB_OFF.ET;
  DB_FF (SET1:= DB_ON.Q, RESET := DB_OFF.Q);
  OUT := DB_FF.Q1;
END_FUNCTION_BLOCK
```

Listing B.10: *ST Model: ST_DEBOUNCE*

```
FUNCTION_BLOCK ST_ASS_DEL
  VAR
    x0 : INT := 2;
    y0 : INT := 1;
  END_VAR

  y0 := y0 + x0;
END_FUNCTION_BLOCK
```

Listing B.11: *ST Model: ST_ASS_DEL*

```
FUNCTION_BLOCK ST_OP_IN_EQ
  VAR
    x0 : BOOL;
    x1 : BOOL := TRUE;
    x2 : BOOL := FALSE;
  END_VAR

  x0 := x1 = x2;
  x0 := x1 <> x2;
END_FUNCTION_BLOCK
```

Listing B.12: *ST Model: ST_OP_IN_EQ*

```
FUNCTION_BLOCK ST_LOOP_FOOT
  VAR_OUTPUT
    y : INT;
  END_VAR
  VAR
    x0 : INT := 0;
    x1 : INT := 1;
    x2 : INT := 2;
    i : INT;
    i0 : INT := 0;
    i1 : INT := 10;
  END_VAR
```

```
13
14    i := i0;
15    REPEAT
16      y := x0;
17      i := i + x2;
18    UNTIL i > i1
19    END_REPEAT;
20    y := x1;
21  END_FUNCTION_BLOCK
```

Listing B.13: *ST Model: ST_LOOP_FOOT*

```
1   FUNCTION_BLOCK ST_LOOP_HEAD
2     VAR_OUTPUT
3       y : INT;
4     END_VAR
5     VAR
6       x1 : INT := 1;
7       x2 : INT := 2;
8       i : INT;
9       i0 : INT := 0;
10      i1 : INT := 10;
11    END_VAR
12
13    i := i0;
14    WHILE i <= i1 DO
15      y := i;
16      i := i + x2;
17    END_WHILE;
18    y := x1;
19  END_FUNCTION_BLOCK
```

Listing B.14: *ST Model: ST_LOOP_HEAD*

```
1   FUNCTION_BLOCK ST_ASS_IMM1
2     VAR
3       x : INT := 2;
4       y : INT := 2;
5       x0 : INT := 2;
6       y0 : INT;
7       y1 : INT;
8       x1 : INT;
9     END_VAR
10
11    y := x;
12    y0 := x0;
13    y1 := x1;
14  END_FUNCTION_BLOCK
```

Listing B.15: *ST Model: ST_ASS_IMM1*

```
1   FUNCTION_BLOCK ST_ASS_IMM2
2     VAR
3       x0 : INT := 2;
4       x1 : INT := 2;
5       x2 : INT := 2;
6       y0 : INT;
7       y1 : INT;
8       y2 : INT := 2;
9     END_VAR
10
11    y0 := x0;
12    y1 := x1;
13    y0 := x2;
14
```

```
  y2 := x0;
  y2 := x1;

  y0 := x0;
  x0 := y0 + x1;
END_FUNCTION_BLOCK
```

Listing B.16: *ST Model: ST_ASS_IMM2*

```
FUNCTION ST_ASS_IMM_OUT : BOOL
  VAR_INPUT
    x : INT;
  END_VAR
  VAR_OUTPUT
    y: INT;
  END_VAR

  y := x;
  ST_ASS_IMM_OUT := TRUE;
END_FUNCTION
```

Listing B.17: *ST Model: ST_ASS_IMM_OUT*

```
FUNCTION_BLOCK ST_ASS_IMM3
  VAR
    y1: INT;
    y2: BOOL;
  END_VAR

  ST_ASS_IMM_OUT(x := 4, y => y1, ST_ASS_IMM_OUT => y2);
END_FUNCTION_BLOCK
```

Listing B.18: *ST Model: ST_ASS_IMM3*

```
FUNCTION ST_LEFT1 : BOOL
  VAR_INPUT
  xSENSOR_L : BOOL;
  xSENSOR_R : BOOL;
  END_VAR

  ST_LEFT1 := xSENSOR_L AND NOT xSENSOR_R;
END_FUNCTION
```

Listing B.19: *ST Model: ST_LEFT1*

```
FUNCTION_BLOCK ST_OP_NUM_REL
  VAR
    x0 : BOOL;
    x1 : INT := 1;
    x2 : INT := 2;
  END_VAR

  x0 := x1 < x2;
  x0 := x1 <= x2;
  x0 := x1 > x2;
  x0 := x1 >= x2;
END_FUNCTION_BLOCK
```

Listing B.20: *ST Model: ST_OP_NUM_REL*

```
FUNCTION_BLOCK ST_TOF
  VAR_INPUT
    CLK: TIME;
    IN: BOOL;
```

```
    PT: TIME;
  END_VAR
  VAR_OUTPUT
    Q1: BOOL := FALSE;
    ET: TIME;
  END_VAR
  VAR
    ETTIME: TIME;
    TSTART: TIME;
    LASTIN: BOOL;
    Q1_Temp1: BOOL;
  END_VAR

  IF (IN <> LASTIN) THEN
    LASTIN := IN;
    IF IN THEN
      TSTART := CLK;
    ELSE
      TSTART := T#0MS;
  END_IF;
    Q1_Temp1 := TRUE;
    Q1 := Q1_Temp1;
    ET := T#0MS;
  ELSE
  IF ((NOT IN) AND Q1_Temp1) THEN
      ETTIME := CLK - TSTART;
      IF (ETTIME < PT) THEN
        ET := ETTIME;
      ELSE
        Q1_Temp1 := FALSE;
        Q1 := Q1_Temp1;
        ET := PT;
    END_IF;
  END_IF;
END_IF;
END_FUNCTION_BLOCK
```

Listing B.21: *ST Model:* `ST_TOF`

```
FUNCTION_BLOCK ST_TON
  VAR_INPUT
    CLK: TIME;
    IN: BOOL;
    PT: TIME;
  END_VAR
  VAR_OUTPUT
  Q1: BOOL := FALSE;
  ET: TIME;
  END_VAR
  VAR
    ETTIME: TIME;
    TSTART: TIME;
    LASTIN: BOOL;
    Q1_Temp1: BOOL;
  END_VAR

  IF (IN <> LASTIN) THEN
    LASTIN := IN;
    IF IN THEN
      TSTART := CLK;
    ELSE
      TSTART := T#0MS;
  END_IF;
    Q1_Temp1 := FALSE;
    Q1 := Q1_Temp1;
    ET := T#0MS;
```

```
28    ELSE
29      IF (IN AND (NOT Q1_Temp1)) THEN
30        ETTIME := CLK - TSTART;
31        IF (ETTIME < PT) THEN
32          ET := ETTIME;
33        ELSE
34          Q1_Temp1 := TRUE;
35          Q1 := Q1_Temp1;
36          ET := PT;
37        END_IF;
38      END_IF;
39    END_IF;
40  END_FUNCTION_BLOCK
```

Listing B.22: *ST Model: ST_TON*

```
1   FUNCTION_BLOCK ST_RS
2     VAR_INPUT
3       SET: BOOL;
4       RESET1: BOOL;
5     END_VAR
6     VAR_OUTPUT
7       Q1: BOOL;
8     END_VAR
9     VAR
10      Q1_Tmp: BOOL;
11    END_VAR
12
13    Q1_Tmp := (SET OR Q1_Tmp) AND (NOT RESET1);
14    Q1 := Q1_Tmp;
15  END_FUNCTION_BLOCK
```

Listing B.23: *ST Model: ST_RS*

```
1   FUNCTION ST_RIGHT1 : BOOL
2     VAR_INPUT
3       xSENSOR_L : BOOL;
4       xSENSOR_R : BOOL;
5     END_VAR
6
7     ST_RIGHT1 := NOT xSENSOR_L AND xSENSOR_R;
8   END_FUNCTION
```

Listing B.24: *ST Model: ST_RIGHT1*

```
1   FUNCTION_BLOCK ST_SR
2     VAR_INPUT
3       SET1 : BOOL;
4       RESET : BOOL;
5     END_VAR
6     VAR_OUTPUT
7       Q1 : BOOL;
8     END_VAR
9     VAR
10      Q1_Tmp : BOOL;
11    END_VAR
12
13    Q1_Tmp := SET1 OR (Q1_Tmp AND (NOT RESET));
14    Q1 := Q1_Tmp;
15  END_FUNCTION_BLOCK
```

Listing B.25: *ST Model: ST_SR*

```
FUNCTION ST_SIMPLE_FUN : REAL
  VAR_INPUT
    A1 : REAL;
    B1 : REAL;
    C1 : REAL := 1.0;
  END_VAR
  VAR_IN_OUT
    COUNT : INT;
  END_VAR
  VAR
    COUNTP1 : INT;
  END_VAR

  COUNTP1 := COUNT + 1;
  COUNT := COUNTP1;
  ST_SIMPLE_FUN := (A1 * B1) / C1;
END_FUNCTION
```

Listing B.26: *ST Model: ST_SIMPLE_FUN*

```
PROGRAM ST_SIMPLE_PRG1
  VAR_INPUT
    PRG_IN : BOOL;
    PRG_A : REAL;
    PRG_B : REAL;
    PRG_C : REAL;
  END_VAR
  VAR_OUTPUT
    PRG_OUT1 : BOOL;
    PRG_OUT2 : REAL := 1.0;
    PRG_ET_OFF : TIME;
  END_VAR
  VAR
    DEBOUNCE_01 : ST_DEBOUNCE;
    PRG_COUNT : INT := 4;
  END_VAR

  DEBOUNCE_01(IN := PRG_IN, DB_TIME := T#2000MS, OUT => PRG_OUT1, ET_OFF
      => PRG_ET_OFF);
  ST_SIMPLE_FUN(A1 := PRG_A + 2.0, B1 := PRG_B, C1 := PRG_C, COUNT :=
     PRG_COUNT, ST_SIMPLE_FUN => PRG_OUT2);
END_PROGRAM
```

Listing B.27: *ST Model: ST_SIMPLE_PRG1*

```
PROGRAM ST_TANK_CTRL
  VAR
    S1 : BOOL;
    B1 : BOOL;
    B2 : BOOL;
    P1 : BOOL;
    M1 : BOOL;
    M2 : BOOL;
  END_VAR

  P1 := S1;
  IF NOT B1 THEN
    M1 := TRUE;
    M2 := TRUE;
  END_IF;
  IF B2 OR NOT S1 THEN
    M1 := FALSE;
    M2 := FALSE;
  END_IF;
```

```
END_PROGRAM
```

Listing B.28: *ST Model: ST_TANK_CTRL*

```
PROGRAM ST_TRACK_CORR
  VAR_INPUT
    B1 : BOOL;
    B2 : BOOL;
    B3 : BOOL;
  END_VAR
  VAR_OUTPUT
    P1_Alarm : BOOL;
    K1_Left : BOOL;
    K2_Right : BOOL;
  END_VAR
  VAR
    Tmp1 : BOOL;
    Tmp2 : BOOL;
    Tmp3 : BOOL;
  END_VAR

  ST_ALARM(xSENSOR_L := B1, xSENSOR_M := B2, xSENSOR_R := B3, ST_ALARM
      => Tmp1);
  P1_Alarm := Tmp1;
  ST_LEFT1(xSENSOR_L := B1, xSENSOR_R := B3, ST_LEFT1 => Tmp2);
  K1_Left := Tmp2;
  ST_RIGHT1(xSENSOR_L := B1, xSENSOR_R := B3, ST_RIGHT1 => Tmp3);
  K2_Right := Tmp3;
END_PROGRAM
```

Listing B.29: *ST Model: ST_TRACK_CORR*

```
PROGRAM ST_TWO_PCTRL
  VAR
    usiS : UINT := 150;
    usiH : UINT := 200;
    usiOn : UINT;
    usiOff : UINT;
    xOut : BOOL;
  END_VAR
END_PROGRAM

  IF usiS > 200 OR usiS < 100 THEN
    usiS := 150;
  END_IF;
  usiOn := usiS - 25;
  usiOff := usiS + 25;
  IF usiH > usiOff THEN
    xOut := FALSE;
  END_IF;
  IF usiH < usiOn THEN
    xOut := TRUE;
  END_IF;
END_PROGRAM
```

Listing B.30: *ST Model: ST_TWO_PCTRL (USINT data type designed as UINT)*

Appendix C

Resulting ST-Based Quartz Models

```
module ST_TWO_OF_THREE(
    event bool ?EI,
    event bool !EO,
    bool ?xB1_Temp,
    bool ?xB2_Temp,
    bool ?xB3_Temp,
    bool !xP1_Temp){

    loop{
        immediate await(EI);

        xP1_Temp = ((xB1_Temp&xB2_Temp)|(xB1_Temp&xB3_Temp)|(xB2_Temp&
            xB3_Temp));

        emit(EO); pause;
    }
}
```

Listing C.1: *Quartz Model: ST_TWO_OF_THREE*

```
module ST_ALARM(
    bool ?xSENSOR_L,
    bool ?xSENSOR_M,
    bool ?xSENSOR_R,
    bool !ST_ALARM){

    ST_ALARM = (!(xSENSOR_L)&!(xSENSOR_M)&!(xSENSOR_R))|(xSENSOR_L&
        xSENSOR_R);
}
```

Listing C.2: *Quartz Model: ST_ALARM*

```
module ST_SCALE(
    int{32768} ?iX,
    real ?rY_MAX,
    real ?rY_MIN,
    real !ST_SCALE){

    ST_SCALE = (rY_MAX - rY_MIN) / 32760.0 * iX + rY_MIN;
}
```

Listing C.3: *Quartz Model: ST_SCALE*

```
import ST_SCALE.*;

module ST_AVAL_PROC(
    event bool ?EI,
    event bool !EO){

    real rPressure;
    int{32768} iPressure_Per;
    real ST_SCALE_l1;


    loop{
        immediate await(EI);

        ST_SCALE(iPressure_Per, 10.0, 6.0, ST_SCALE_l1);
        rPressure = ST_SCALE_l1;

        emit(EO); pause;
    }
}
```

Listing C.4: *Quartz Model: ST_AVAL_PROC*

```
module ST_OP_ARITH(
    event bool ?EI,
    event bool !EO){

    real x01;
    real x02;
    real x03;
    real x04;
    real x05;
    real x06;
    real x1;
    real x2;
    int{32768} x3;
    int{32768} x4;

    x1 = 1.0;
    x2 = 2.0;
    x3 = 1;
    x4 = 2;
    pause;

    loop{
        immediate await(EI);

        x01 = x1 + x2;
        x02 = x1 - x2;
        x03 = x1 * x2;
        x04 = x1 / x2;
        x05 = exp(x1, x2);
        x06 = x3 % x4;

        emit(EO); pause;
    }
}
```

Listing C.5: *Quartz Model: ST_OP_ARITH*

```
module ST_OP_BOOL(
    event bool ?EI,
    event bool !EO){

    bool x01;
```

```
    bool x02;
    bool x03;
    bool x04;
    bool x1;
    bool x2;

    x1 = true;
    x2 = false;
    pause;

    loop{
        immediate await(EI);

        x01 = !(x1);
        x02 = x1 & x2;
        x03 = x1 | x2;
        x04 = x1 ^ x2;

        emit(EO); pause;
    }
}
```

Listing C.6: *Quartz Model:* `ST_OP_BOOL`

```
module ST_COMPENS(
    event bool ?EI,
    event bool !EO){

    real rQ;
    bool xQ1;
    bool xQ2;
    bool xQ3;

    rQ = 1500.0;
    pause;

    loop{
        immediate await(EI);

        if(rQ < 1000.0){
            xQ3 = false;
            xQ2 = false;
            xQ1 = false;
        }

        if(rQ >= 1000.0){
            xQ3 = false;
            xQ2 = false;
            xQ1 = true;
        }

        if(rQ >= 2000.0){
            xQ3 = false;
            xQ2 = true;
            xQ1 = true;
        }

        if(rQ > 3000.0){
            xQ3 = true;
            xQ2 = true;
            xQ1 = true;
        }

        emit(EO); pause;
    }
```

```
}
```

Listing C.7: *Quartz Model:* `ST_COMPENS`

```
module ST_COND(
    event bool ?EI,
    event bool !EO){

    bool x0;
    bool x1;
    bool x2;

    x1 = true;
    x2 = true;
    pause;

    loop{
        immediate await(EI);

        if(x1){
            x2 = true;
        }

        if(x1){
            x0 = true;
        }else{
            x0 = false;
        }

        emit(EO); pause;
    }
}
```

Listing C.8: *Quartz Model:* `ST_COND`

```
module ST_DATATYPES(
    event bool ?EI,
    event bool !EO){

    bool A1;
    bool A2;
    bv{16} A3;
    bv{32} A4;
    int{32768} A5;
    int{32768} A6;
    int{2147483648} A7;
    int{2147483648} A8;
    nat{65536} A9;
    nat{65536} A10;
    nat{4294967296} A11;
    nat{4294967296} A12;
    real A13;
    real A14;
    nat A15;
    nat A16;
    [3]bool A17;
    [3]bv{16} A18;
    [3]bv{32} A19;
    [3]int{32768} A20;
    [3]int{2147483648} A21;
    [3]nat{65536} A22;
    [3]nat{4294967296} A23;
    [3]real A24;
    [3]nat A25;

```

```
31      A2 = true;
32      A6 = 2;
33      A8 = 2;
34      A10 = 2;
35      A12 = 2;
36      A14 = 1.23;
37      A16 = 5000;
38      pause;
39
40      loop{
41          immediate await(EI);
42          emit(EO); pause;
43      }
44  }
```

Listing C.9: *Quartz Model: ST_DATATYPES*

```
1   import TON.*;
2   import SR.*;
3
4   module ST_DEBOUNCE(
5       event bool ?EI,
6       event bool !EO,
7       nat ?CLK,
8       bool ?IN,
9       nat ?DB_TIME,
10      bool !OUT,
11      nat    !ET_OFF){
12
13      event bool DB_ON_EI;
14      event bool DB_ON_EO;
15      event bool DB_OFF_EI;
16      event bool DB_OFF_EO;
17      event bool DB_FF_EI;
18      event bool DB_FF_EO;
19
20      bool DB_ON_Q;
21      bool DB_OFF_Q;
22      bool DB_FF_Q1;
23      nat DB_ON_ET;
24      nat DB_OFF_ET;
25
26      loop{
27          immediate await(EI);
28
29          emit(DB_ON_EI);
30          immediate await(DB_ON_EO);
31
32          emit(DB_OFF_EI);
33          immediate await(DB_OFF_EO);
34
35          ET_OFF = DB_OFF_ET;
36
37          emit(DB_FF_EI);
38          immediate await(DB_FF_EO);
39
40          OUT = DB_FF_Q1;
41
42          emit(EO);
43          pause;
44      }
45      || DB_ON:TON(DB_ON_EI, DB_ON_EO, CLK,
46          IN, DB_TIME,
47          DB_ON_Q, DB_ON_ET);
48      || DB_OFF:TON(DB_OFF_EI, DB_OFF_EO, CLK,
49          !(IN),
```

```
        DB_TIME,
        DB_OFF_Q, DB_OFF_ET);
    || DB_FF:SR(DB_FF_EI, DB_FF_EO,
        DB_ON_Q,
        DB_OFF_Q, DB_FF_Q1);
}
```

Listing C.10: *Quartz Model:* `ST_DEBOUNCE`

```
module ST_ASS_DEL(
    event bool ?EI,
    event bool !EO){

    int{32768} x0;
    int{32768} y0;

    x0 = 2;
    y0 = 1;
    pause;

    loop{
        immediate await(EI);

        next(y0) = y0 + x0;
        pause;

        emit(EO); pause;
    }
}
```

Listing C.11: *Quartz Model:* `ST_ASS_DEL`

```
module ST_OP_IN_EQ(
    event bool ?EI,
    event bool !EO){

    bool x0;
  bool x1;
    bool x2;

    x1 = true;
    x2 = false;
    pause;

    loop{
        immediate await(EI);

        x0 = x1 == x2;
        pause;
        x0 = x1 != x2;

        emit(EO); pause;
    }
}
```

Listing C.12: *Quartz Model:* `ST_OP_IN_EQ`

```
module ST_LOOP_FOOT(
    event bool ?EI,
    event bool !EO,
    int{32768} !y){

    int{32768} x0;
    int{32768} x1;
    int{32768} x2;
```

```
    int{32768} i;
    int{32768} i0;
    int{32768} i1;

    x0 = 0;
    x1 = 1;
    x2 = 2;
    i0 = 0;
    i1 = 10;
    pause;

    loop{
        immediate await(EI);

        i = i0;
        do{
            y = x0;
            next(i) = i + x2;
            pause;
        }while(!(i > i1));
        y = x1;

        emit(EO); pause;
    }
}
```

Listing C.13: *Quartz Model: ST_LOOP_FOOT*

```
module ST_LOOP_HEAD(
    event bool ?EI,
    event bool !EO,
    int{32768} !y){

    int{32768} x1;
    int{32768} x2;
    int{32768} i;
    int{32768} i0;
    int{32768} i1;

    x1 = 1;
    x2 = 2;
    i0 = 0;
    i1 = 10;
    pause;

    loop{
        immediate await(EI);

        i = i0;
        while(i <= i1){
            y = i;
            next(i) = i + x2;
            pause;
        }
        y = x1;

        emit(EO); pause;
    }
}
```

Listing C.14: *Quartz Model: ST_LOOP_HEAD*

```
module ST_ASS_IMM1(
    event bool ?EI,
    event bool !EO){
```

```
4
5       int{32768} x;
6       int{32768} y;
7       int{32768} x0;
8       int{32768} y0;
9       int{32768} y1;
10      int{32768} x1;
11
12      x = 2;
13      y = 2;
14      x0 = 2;
15      pause;
16
17      loop{
18          immediate await(EI);
19
20          y = x;
21          y0 = x0;
22          y1 = x1;
23
24          emit(EO); pause;
25      }
26  }
```

Listing C.15: *Quartz Model:* `ST_ASS_IMM1`

```
1   module ST_ASS_IMM2(
2       event bool ?EI,
3       event bool !EO){
4
5       int{32768} x0;
6       int{32768} x1;
7       int{32768} x2;
8       int{32768} y0;
9       int{32768} y1;
10      int{32768} y2;
11
12      x0 = 2;
13      x1 = 2;
14      x2 = 2;
15      y2 = 2;
16      pause;
17
18      loop{
19          immediate await(EI);
20
21          y0 = x0;
22          y1 = x1;
23          pause;
24          y0 = x2;
25
26          y2 = x0;
27          pause;
28          y2 = x1;
29
30          y0 = x0;
31          pause;
32          x0 = y0 + x1;
33
34          emit(EO); pause;
35      }
36  }
```

Listing C.16: *Quartz Model:* `ST_ASS_IMM2`

```
module ST_ASS_IMM_OUT(
    int{32768} ?x,
    int{32768} !y,
    bool !ST_ASS_IMM_OUT){

    y = x;
    ST_ASS_IMM_OUT = true;
}
```

Listing C.17: *Quartz Model: ST_ASS_IMM_OUT*

```
import ST_ASS_IMM_OUT.*;

module ST_ASS_IMM3(
    event bool ?EI,
    event bool !EO){

    int{32768} y1;
    bool y2;
    int{32768} ST_ASS_IMM_OUT_y_l1;
    bool ST_ASS_IMM_OUT_l1;

    loop{
        immediate await(EI);

        ST_ASS_IMM_OUT(4, ST_ASS_IMM_OUT_y_l1, ST_ASS_IMM_OUT_l1);
        y1 = ST_ASS_IMM_OUT_y_l1;
        y2 = ST_ASS_IMM_OUT_l1;

        emit(EO); pause;
    }
}
```

Listing C.18: *Quartz Model: ST_ASS_IMM3*

```
module ST_LEFT1(
    bool ?xSENSOR_L,
    bool ?xSENSOR_R,
    bool !ST_LEFT1){

    ST_LEFT1 = (xSENSOR_L & !(xSENSOR_R));
}
```

Listing C.19: *Quartz Model: ST_LEFT1*

```
module ST_OP_NUM_REL(
    event bool ?EI,
    event bool !EO){

    bool x0;
  int{32768} x1;
    int{32768} x2;

    x1 = 1;
    x2 = 2;
    pause;

    loop{
        immediate await(EI);

        x0 = x1 < x2;
        pause;
        x0 = x1 <= x2;
        pause;
        x0 = x1 > x2;
```

```
        pause;
        x0 = x1 >= x2;

        emit(EO); pause;
    }
}
```

Listing C.20: *Quartz Model: ST_OP_NUM_REL*

```
module ST_TOF(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
    bool ?IN,
    nat ?PT,
    bool !Q1,
    nat !ET){

    nat ETTIME;
    nat TSTART;
    bool LASTIN;
    bool Q1_Temp1;

    Q1 = false;
    pause;

    loop{
        immediate await(EI);

        if(IN != LASTIN){
            next(LASTIN) = IN;
            pause;
            if(IN){
                TSTART = CLK;
            }else{
                TSTART = 0;
            }
            Q1_Temp1 = true;
            Q1 = Q1_Temp1;
            ET = 0;
        }
        else{
            if(!(IN) & Q1_Temp1){
                ETTIME = CLK - TSTART;
                if(ETTIME < PT){
                    ET = ETTIME;
                }else{
                    next(Q1_Temp1) = false;
                    pause;
                    Q1 = Q1_Temp1;
                    ET = PT;
                }
            }
        }

        emit(EO); pause;
    }
}
```

Listing C.21: *Quartz Model: ST_TOF*

```
module ST_TON(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
```

```
    bool ?IN,
    nat ?PT,
    bool !Q1,
    nat !ET){

    nat ETTIME;
    nat TSTART;
    bool LASTIN;
    bool Q1_Temp1;

    Q1 = false;
    pause;

    loop{
        immediate await(EI);

        if(IN != LASTIN){
            next(LASTIN) = IN;
            pause;
            if(IN){
                TSTART = CLK;
            }else{
                TSTART = 0;
            }
            Q1_Temp1 = false;
            Q1 = Q1_Temp1;
            ET = 0;
        }
        else{
            if(IN & !(Q1_Temp1)){
                ETTIME = CLK - TSTART;
                if(ETTIME < PT){
                    ET = ETTIME;
                }else{
                    next(Q1_Temp1) = true;
                    pause;
                    Q1 = Q1_Temp1;
                    ET = PT;
                }
            }
        }

        emit(EO); pause;
  }
}
```

Listing C.22: *Quartz Model: ST_TON*

```
module ST_RS(
    event bool ?EI,
    event bool !EO,
    bool ?SET,
    bool ?RESET1,
    bool !Q1){

    bool Q1_Tmp;

    loop{
        immediate await(EI);

        next(Q1_Tmp) = (SET | Q1_Tmp) & !(RESET1);
        pause;
        Q1 = Q1_Tmp;

        emit(EO); pause;
    }
```

```
}
```

Listing C.23: *Quartz Model: ST_RS*

```
module ST_RIGHT1(
    bool ?xSENSOR_L,
    bool ?xSENSOR_R,
    bool !ST_RIGHT1){

    ST_RIGHT1 = (!(xSENSOR_L) & xSENSOR_R);
}
```

Listing C.24: *Quartz Model: ST_RIGHT1*

```
module ST_SR(
    event bool ?EI,
    event bool !EO,
    bool ?SET1,
    bool ?RESET,
    bool !Q1){

    bool Q1_Tmp;

    loop{
        immediate await(EI);

        next(Q1_Tmp) = SET1 | (Q1_Tmp & !(RESET));
        pause;
        Q1 = Q1_Tmp;

        emit(EO); pause;
    }
}
```

Listing C.25: *Quartz Model: ST_SR*

```
module ST_SIMPLE_FUN(
    real ?A1,
    real ?B1,
    real ?C1,
    int{32768} COUNT,
    real !ST_SIMPLE_FUN){

    int{32768} COUNTP1;

    COUNTP1 = COUNT+1;
    pause;
    COUNT = COUNTP1;
    ST_SIMPLE_FUN = (A1*B1)/C1;
}
```

Listing C.26: *Quartz Model: ST_SIMPLE_FUN*

```
import ST_DEBOUNCE.*;
import ST_SIMPLE_FUN.*;

module ST_SIMPLE_PRG1(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
    bool ?PRG_IN,
    real ?PRG_A,
    real ?PRG_B,
    real ?PRG_C,
```

```
    bool !PRG_OUT1,
    nat !PRG_ET_OFF,
    real !PRG_OUT2){

    int{32768} PRG_COUNT;

    event bool DEBOUNCE_01_EI;
    event bool DEBOUNCE_01_EO;

    bool DEBOUNCE_01_OUT;
    nat DEBOUNCE_01_ET_OFF;

    real ST_SIMPLE_FUN_l1;

    PRG_COUNT = 4;
    pause;

    loop{
        immediate await(EI);

        emit(DEBOUNCE_01_EI);
        immediate await(DEBOUNCE_01_EO);

        PRG_OUT1 = DEBOUNCE_01_OUT;
        PRG_ET_OFF = DEBOUNCE_01_ET_OFF;

        ST_SIMPLE_FUN(
            PRG_A + 2.0,
            PRG_B, PRG_C,
            PRG_COUNT, ST_SIMPLE_FUN_l1);

        PRG_OUT2 = ST_SIMPLE_FUN_l1;

        emit(EO);
        pause;
    }
    || DEBOUNCE_01:ST_DEBOUNCE(DEBOUNCE_01_EI, DEBOUNCE_01_EO, CLK,
        PRG_IN,
        2000,
        DEBOUNCE_01_OUT, DEBOUNCE_01_ET_OFF);
}
```

Listing C.27: *Quartz Model: ST_SIMPLE_PRG1*

```
module ST_TANK_CTRL(
    event bool ?EI,
    event bool !EO){

    bool S1;
    bool B1;
    bool B2;
    bool P1;
    bool M1;
    bool M2;

    loop{
        immediate await(EI);

        P1 = S1;
        if(!(B1)){
            M1 = true;
            M2 = true;
        }
        if(B2 | !(S1)){
            pause;
            M1 = false;
```

```
            M2 = false;
        }

        emit(EO); pause;
    }
}
```

Listing C.28: *Quartz Model: ST_TANK_CTRL*

```
import ST_ALARM.*;
import ST_LEFT1.*;
import ST_RIGHT1.*;

module ST_TRACK_CORR(
    event bool ?EI,
    event bool !EO,
    bool ?B1,
    bool ?B2,
    bool ?B3,
    bool !P1_Alarm,
    bool !K1_Left,
    bool !K2_Right){

    bool Tmp1;
    bool Tmp2;
    bool Tmp3;

    bool ST_ALARM_l1;
    bool ST_LEFT1_l1;
    bool ST_RIGHT1_l1;

    loop{
        immediate await(EI);

        ST_ALARM(B1, B2, B3,
            ST_ALARM_l1);
        Tmp1 = ST_ALARM_l1;
        P1_Alarm = Tmp1;

        ST_LEFT1(B1, B3,
            ST_LEFT1_l1);
        Tmp2 = ST_LEFT1_l1;
        K1_Left = Tmp2;

        ST_RIGHT1(B1, B3,
            ST_RIGHT1_l1);
        Tmp3 = ST_RIGHT1_l1;
        K2_Right = Tmp3;

        emit(EO); pause;
    }
}
```

Listing C.29: *Quartz Model: ST_TRACK_CORR*

```
module ST_TWO_PCTRL(
    event bool ?EI,
    event bool !EO){

    nat{65536} usiS;
    nat{65536} usiH;
    nat{65536} usiOn;
    nat{65536} usiOff;
    bool xOut;

```

```
11      usiS = 150;
12      usiH = 200;
13      pause;
14
15      loop{
16          immediate await(EI);
17
18          if((usiS > 200) | (usiS < 100)){
19              usiS = 150;
20          }
21          usiOn = usiS - 25;
22          usiOff = usiS + 25;
23
24          if(usiH > usiOff){
25              xOut = false;
26          }
27          if(usiH < usiOn){
28              xOut = true;
29          }
30
31          emit(EO); pause;
32      }
33  }
```

Listing C.30: *Quartz Model: `ST_TWO_PCTRL` (`USINT` data type designed as `UINT`)*

Appendix D

Resulting SCL Models

```
module ST_TWO_OF_THREE{
  input bool EI;
  output bool EO;
  input bool xB1_Temp;
  input bool xB2_Temp;
  input bool xB3_Temp;
  output bool xP1_Temp;

  loop:
    while(!EI){
      pause;
    }

    xP1_Temp = ((xB1_Temp & xB2_Temp) | (xB1_Temp & xB3_Temp) | (
        xB2_Temp & xB3_Temp));

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.1: *SCL Model:* `ST_TWO_OF_THREE`

```
module ST_ALARM{
  input bool EI;
  output bool EO;
  input bool xSENSOR_L;
  input bool xSENSOR_M;
  input bool xSENSOR_R;
  output bool ST_ALARM;

  loop:
    while(!EI){
      pause;
    }
    ST_ALARM = false;

    ST_ALARM = (!(xSENSOR_L) & !(xSENSOR_M) & !(xSENSOR_R)) | (xSENSOR_L
         & xSENSOR_R);

    EO = true;
    pause;
    EO = false;
    goto loop;
```

```
}
```

Listing D.2: *SCL Model: ST_ALARM*

```
module ST_SCALE{
  input bool EI;
  output bool EO;
  input int iX;
  input float rY_MAX;
  input float rY_MIN;
  output bool ST_SCALE;

  loop:
    while(!EI){
      pause;
    }
    ST_SCALE = false;

    ST_SCALE = (rY_MAX - rY_MIN) / 32760.0 * iX + rY_MIN;

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.3: *SCL Model: ST_SCALE*

```
module ST_OP_ARITH{
  input bool EI;
  output bool EO;
  output int y;
  float x01;
  float x02;
  float x03;
  float x04;
  float x05;
  float x06;
  float x1 = 1.0;
  float x2 = 2.0;
  int x3 = 1;
  int x4 = 2;

  loop:
    while(!EI){
      pause;
    }

    x01 = x1 + x2;
    x02 = x1 - x2;
    x03 = x1 * x2;
    x04 = x1 / x2;
    x06 = x3 % x4;

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.4: *SCL Model: ST_OP_ARITH*

```
module ST_OP_BOOL{
  input bool EI;
  output bool EO;
```

```
  output int y;
  bool x01;
  bool x02;
  bool x03;
  bool x04;
  bool x1 = true;
  bool x2 = false;

  loop:
    while(!EI){
      pause;
    }

    x01 = !(x1);
    x02 = x1 & x2;
    x03 = x1 | x2;
    x04 = x1 ^ x2;

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.5: *SCL Model:* `ST_OP_BOOL`

```
module ST_COMPENS{
  input bool EI;
  output bool EO;
  float rQ = 1500.0;
  bool xQ1;
  bool xQ2;
  bool xQ3;

  loop:
    while(!EI){
      pause;
    }

    if(rQ < 1000.0){
      xQ3 = false;
      xQ2 = false;
      xQ1 = false;
    }
    if(rQ >= 1000.0){
      xQ3 = false;
      xQ2 = false;
      xQ1 = true;
    }
    if(rQ >= 2000.0){
      xQ3 = false;
      xQ2 = true;
      xQ1 = true;
    }
    if(rQ > 3000.0){
      xQ3 = true;
      xQ2 = true;
      xQ1 = true;
    }

    EO = true;
    pause;
    EO = false;
  goto loop;
```

```
}
```

Listing D.6: *SCL Model:* `ST_COMPENS`

```
module ST_COND{
  input bool EI;
  output bool EO;
  bool x0;
  bool x1 = true;
  bool x2 = true;

  loop:
    while(!EI){
      pause;
    }

    if(x1){
      x2 = true;
    }
    if(x1){
      x0 = true;
    }else{
      x0 = true;
    }

    EO = true;
    pause;
    EO = false;
  goto loop;
}
```

Listing D.7: *SCL Model:* `ST_COND`

```
module ST_DATATYPES{
  input bool EI;
  output bool EO;
  bool A1;
  bool A2 = true;
  int A5;
  int A6 = 2;
  int A7;
  int A8 = 2;
  int A9;
  int A10 = 2;
  float A13;
  float A14 = 1.23;
  int A15;
  int A16 = 5000;
  bool A17[3];
  int A20[3];
  int A21[3];
  int A22[3];
  float A24[3];
  int A25[3];

  loop:
    while(!EI){
      pause;
    }

    EO = true;
    pause;
    EO = false;
  goto loop;
```

```
}
```

Listing D.8: *SCL Model:* `ST_DATATYPES`

```
module ST_ASS_DEL{
  input bool EI;
  output bool EO;
  int x0 = 2;
  int y0 = 1;

  loop:
    while(!EI){
      pause;
    }

    y0 = y0 + x0;

    EO = true;
    pause;
    EO = false;
  goto loop;
}
```

Listing D.9: *SCL Model:* `ST_ASS_DEL`

```
module ST_OP_IN_EQ{
  input bool EI;
  output bool EO;
  output int y;
  bool x0;
  bool x1 = true;
  bool x2 = false;

  loop:
    while(!EI){
      pause;
    }

    x0 = x1 == x2;
    x0 = x1 != x2;

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.10: *SCL Model:* `ST_OP_IN_EQ`

```
module ST_LOOP_FOOT{
  input bool EI;
  output bool EO;
  output int y;
  int x0 = 0;
  int x1 = 1;
  int x2 = 2;
  int i;
  int i0 = 0;
  int i1 = 10;

  loop:
    while(!EI){
      pause;
    }

```

```
17    i = i0;
18    do:
19      y = x0;
20      i = i + x2;
21      pause;
22    if(!(i > i1)){
23      goto do;
24    }
25    y = x1;
26
27    E0 = true;
28    pause;
29    E0 = false;
30    goto loop;
31  }
```

Listing D.11: *SCL Model:* `ST_LOOP_FOOT`

```
1   module ST_LOOP_HEAD{
2     input bool EI;
3     output bool E0;
4     output int y;
5     int x1 = 1;
6     int x2 = 2;
7     int i;
8     int i0 = 0;
9     int i1 = 10;
10
11    loop:
12      while(!EI){
13        pause;
14      }
15
16      i = i0;
17      while(i <= i1){
18        y = i;
19        i = i + x2;
20        pause;
21      }
22      y = x1;
23
24      E0 = true;
25      pause;
26      E0 = false;
27      goto loop;
28  }
```

Listing D.12: *SCL Model:* `ST_LOOP_HEAD`

```
1   module ST_ASS_IMM1{
2     input bool EI;
3     output bool E0;
4     int x = 2;
5     int y = 2;
6     int x0 = 2;
7     int y0;
8     int y1;
9     int x1;
10
11    loop:
12      while(!EI){
13        pause;
14      }
15
16      y = x;
```

```
    y0 = x0;
    y1 = x1;

    EO = true;
    pause;
    EO = false;
  goto loop;
}
```

Listing D.13: *SCL Model: ST_ASS_IMM1*

```
module ST_ASS_IMM2{
  input bool EI;
  output bool EO;
  int x0 = 2;
  int x1 = 2;
  int x2 = 2;
  int y0;
  int y1;
  int y2 = 2;

  loop:
    while(!EI){
      pause;
    }

    y0 = x0;
    y1 = x1;
    y0 = x2;

    y2 = x0;
    y2 = x1;

    y0 = x0;
    x0 = y0 + x1;

    EO = true;
    pause;
    EO = false;
  goto loop;
}
```

Listing D.14: *SCL Model: ST_ASS_IMM2*

```
module ST_ASS_IMM_OUT{
  input bool EI;
  output bool EO;
  input int x;
  output int y;
  output bool ST_ASS_IMM_OUT;

  loop:
    while(!EI){
      pause;
    }
    y = 0;
    ST_ASS_IMM_OUT = false;

    y = x;
    ST_ASS_IMM_OUT = true;

    EO = true;
    pause;
    EO = false;
  goto loop;
```

```
}
```

Listing D.15: *SCL Model:* `ST_ASS_IMM_OUT`

```
module ST_LEFT1{
  input bool EI;
  output bool EO;
  input bool xSENSOR_L;
  input bool xSENSOR_R;
  output bool ST_LEFT1;

  loop:
    while(!EI){
      pause;
    }
    ST_LEFT1 = false;

    ST_LEFT1 = ((xSENSOR_L) & (!(xSENSOR_R)));

    EO = true;
    pause;
    EO = false;
  goto loop;
}
```

Listing D.16: *SCL Model:* `ST_LEFT1`

```
module ST_OP_NUM_REL{
  input bool EI;
  output bool EO;
  output int y;
  bool x0;
  int x1 = 1;
  int x2 = 2;

  loop:
    while(!EI){
      pause;
    }

    x0 = x1 < x2;
    x0 = x1 <= x2;
    x0 = x1 > x2;
    x0 = x1 >= x2;

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.17: *SCL Model:* `ST_OP_NUM_REL`

```
module ST_TOF{
  input bool EI;
  output bool EO;
  input int CLK;
  input bool IN;
  input int PT;
  output bool Q1 = false;
  output int ET;
  int ETTIME;
  int TSTART;
  bool LASTIN;
  bool Q1_Temp1;
```

```
13
14    loop:
15      while(!EI){
16        pause;
17      }
18
19      if(IN != LASTIN){
20          LASTIN = IN;
21          if(IN){
22            TSTART = CLK;
23          }else{
24            TSTART = 0;
25          }
26          Q1_Temp1 = true;
27          Q1 = Q1_Temp1;
28          ET = 0;
29      }else{
30        if(!(IN) & Q1_Temp1){
31          ETTIME = CLK - TSTART;
32          if(ETTIME < PT){
33            ET = ETTIME;
34          }else{
35            Q1_Temp1 = false;
36            Q1 = Q1_Temp1;
37            ET = PT;
38          }
39        }
40      }
41
42      EO = true;
43      pause;
44      EO = false;
45      goto loop;
46  }
```

Listing D.18: *SCL Model:* `ST_TOF`

```
1   module ST_TON{
2     input bool EI;
3     output bool EO;
4     input int CLK;
5     input bool IN;
6     input int PT;
7     output bool Q1 = false;
8     output int ET;
9     int ETTIME;
10    int TSTART;
11    bool LASTIN;
12    bool Q1_Temp1;
13
14    loop:
15      while(!EI){
16        pause;
17      }
18
19      if(IN != LASTIN){
20          LASTIN = IN;
21          if(IN){
22            TSTART = CLK;
23          }else{
24            TSTART = 0;
25          }
26          Q1_Temp1 = false;
27          Q1 = Q1_Temp1;
28          ET = 0;
29      }else{
```

```
      if(IN & (!(Q1_Temp1))){
        ETTIME = CLK - TSTART;
        if(ETTIME < PT){
          ET = ETTIME;
        }else{
          Q1_Temp1 = true;
          Q1 = Q1_Temp1;
          ET = PT;
        }
      }
    }

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.19: *SCL Model: ST_TON*

```
module ST_RS{
  input bool EI;
  output bool EO;
  input bool SET;
  input bool RESET1;
  output bool Q1;
  bool Q1_Tmp;

  loop:
    while(!EI){
      pause;
    }

    Q1_Tmp = (SET | Q1_Tmp) & (!(RESET1));
    Q1 = Q1_Tmp;

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.20: *SCL Model: ST_RS*

```
module ST_RIGHT1{
  input bool EI;
  output bool EO;
  input bool xSENSOR_L;
  input bool xSENSOR_R;
  output bool ST_RIGHT1;

  loop:
    while(!EI){
      pause;
    }
    ST_RIGHT1 = false;

    ST_RIGHT1 = (!(xSENSOR_L) & (xSENSOR_R));

    EO = true;
    pause;
    EO = false;
  goto loop;
}
```

Listing D.21: *SCL Model: ST_RIGHT1*

```
module ST_SR{
  input bool EI;
  output bool EO;
  input bool SET1;
  input bool RESET;
  output bool Q1;
  bool Q1_Tmp;

  loop:
    while(!EI){
      pause;
    }

    Q1_Tmp = SET1 | (Q1_Tmp & (!(RESET)));
    Q1 = Q1_Tmp;

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.22: *SCL Model: ST_SR*

```
module ST_SIMPLE_FUN{
  input bool EI;
  output bool EO;
  input float A1;
  input float B1;
  input float C1 = 1.0;
  input output int COUNT;
  output float ST_SIMPLE_FUN;
  int COUNTP1;

  loop:
    while(!EI){
      pause;
    }
    ST_SIMPLE_FUN = false;

    COUNTP1 = COUNT + 1;
    COUNT = COUNTP1;
    ST_SIMPLE_FUN = ((A1 * B1) / C1);

    EO = true;
    pause;
    EO = false;
    goto loop;
}
```

Listing D.23: *SCL Model: ST_SIMPLE_FUN*

```
module ST_TANK_CTRL{
  input bool EI;
  output bool EO;
  bool S1;
  bool B1;
  bool B2;
  bool P1;
  bool M1;
  bool M2;

  loop:
    while(!EI){
      pause;
```

```
14      }
15
16      P1 = S1;
17      if(!(B1)){
18        M1 = true;
19        M2 = true;
20      }
21      if(B2 | !(S1)){
22        M1 = false;
23        M2 = false;
24      }
25
26      EO = true;
27      pause;
28      EO = false;
29      goto loop;
30  }
```

Listing D.24: *SCL Model:* `ST_TANK_CTRL`

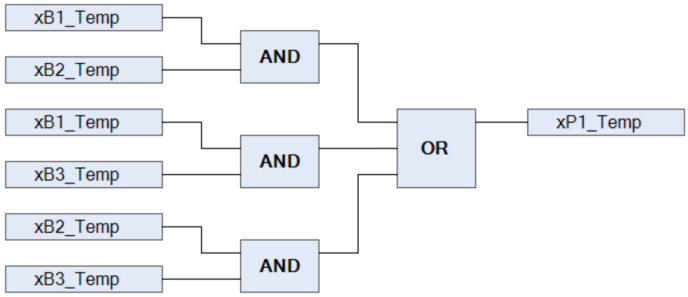
```
1   module ST_TWO_PCTRL{
2     input bool EI;
3     output bool EO;
4     int usiS = 150;
5     int usiH= 200;
6     int usiOn;
7     int usiOff;
8     bool xOut;
9
10    loop:
11      while(!EI){
12        pause;
13      }
14
15      if((usiS > 200)|(usiS < 100)){
16        usiS = 150;
17      }
18      usiOn = usiS - 25;
19      usiOff = usiS + 25;
20      if(usiH > usiOff){
21        xOut = false;
22      }
23      if(usiH < usiOn){
24        xOut = true;
25      }
26
27      EO = true;
28      pause;
29      EO = false;
30    goto loop;
31  }
```

Listing D.25: *SCL Model:* `ST_TWO_PCTRL` *(*`USINT` *data type designed as* `UINT`*)*

Appendix E

FBD Examples

```
1  PROGRAM FBD_TWO_OF_THREE
2    VAR_INPUT
3      xB1_Temp : BOOL;
4      xB2_Temp : BOOL;
5      xB3_Temp : BOOL;
6    END_VAR
7    VAR_OUTPUT
8      xP1_Temp : BOOL;
9    END_VAR
```

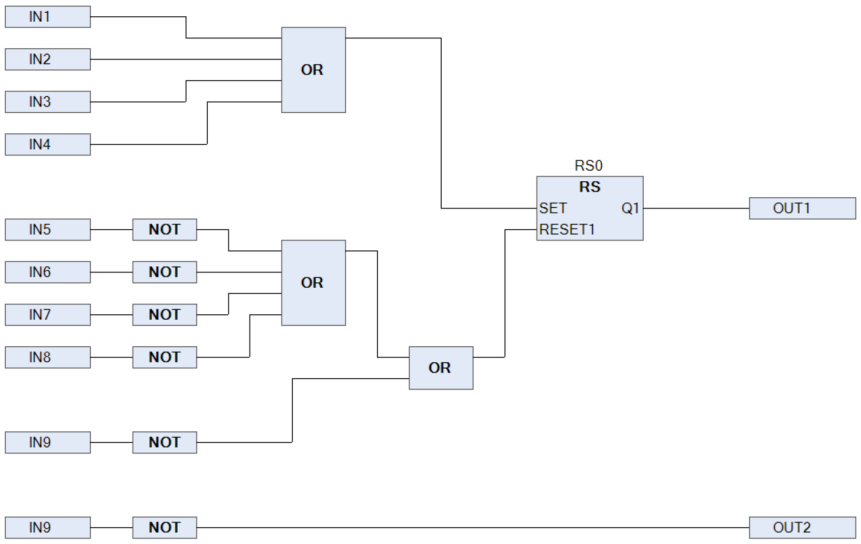



```
10
11 END_PROGRAM
```

Listing E.1: *FBD: FBD_TWO_OF_THREE*

```
1  FUNCTION_BLOCK FBD_AIR_COND_CTRL
2  VAR_INPUT
3    IN1: BOOL;
4    IN2: BOOL;
5    IN3: BOOL;
6    IN4: BOOL;
7    IN5: BOOL;
8    IN6: BOOL;
9    IN7: BOOL;
10   IN8: BOOL;
11   IN9: BOOL;
12 END_VAR
13 VAR_OUTPUT
14   OUT1: BOOL;
15   OUT2: BOOL;
16 END_VAR
```

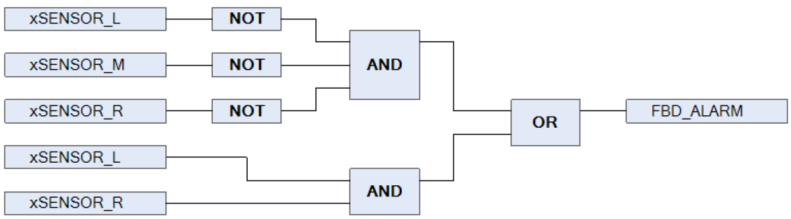
```
17 VAR
18   RS0: RS;
19 END_VAR
```




```
21 END_FUNCTION_BLOCK
```

Listing E.2: *FBD: FBD_AIR_COND_CTRL*

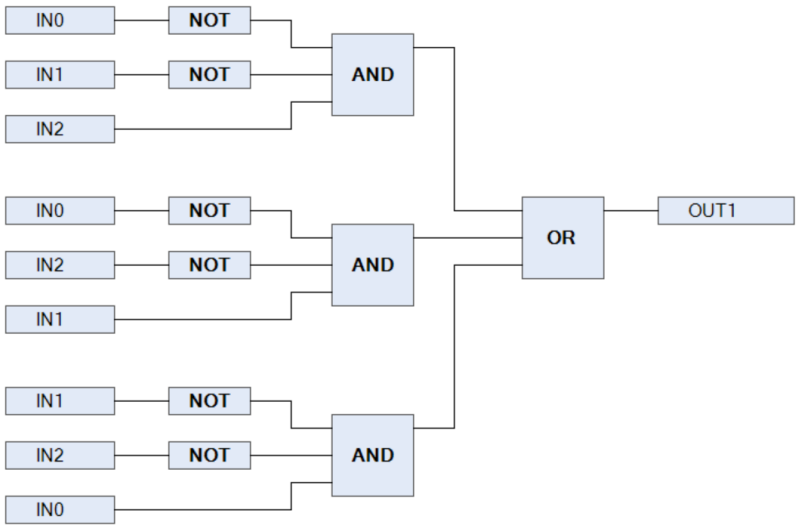
```
1  FUNCTION FBD_ALARM : BOOL
2    VAR_INPUT
3      xSENSOR_L : BOOL;
4      xSENSOR_M : BOOL;
5      xSENSOR_R : BOOL;
6    END_VAR
```




```
8  END_FUNCTION
```

Listing E.3: *FBD: FBD_ALARM*

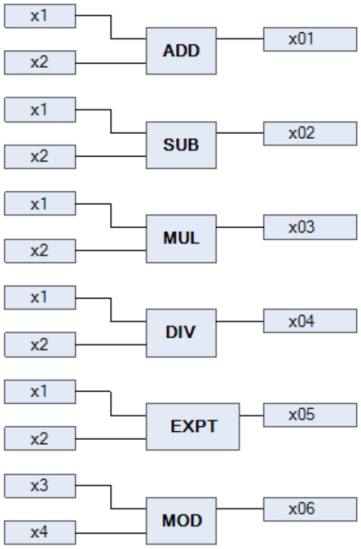
```
1  FUNCTION_BLOCK FBD_ANTIVALENCE
2    VAR_INPUT
3      IN0: BOOL;
4      IN1: BOOL;
5      IN2: BOOL;
6    END_VAR
7    VAR_OUTPUT
8      OUT1: BOOL;
9    END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.4: *FBD: FBD_ANTIVALENCE*

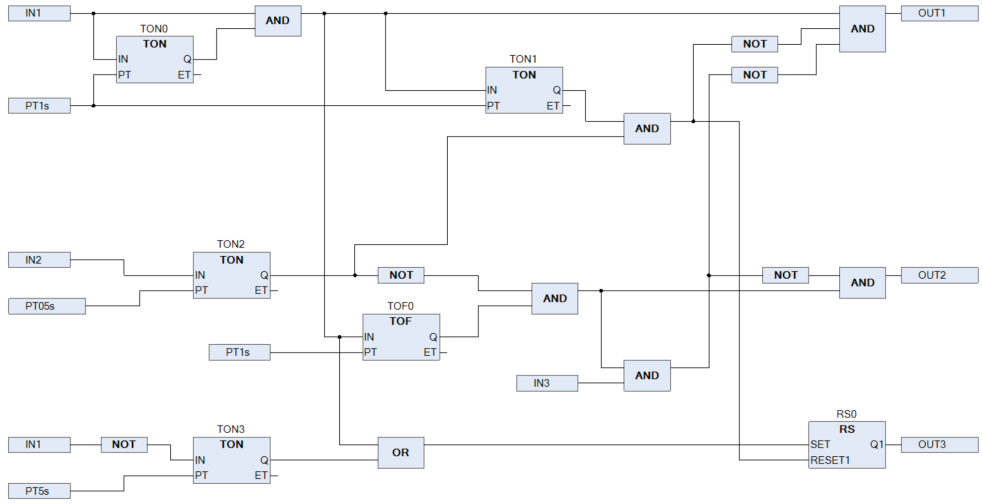
```
FUNCTION_BLOCK FBD_OP_ARITH
  VAR
    x01 : REAL;
    x02 : REAL;
    x03 : REAL;
    x04 : REAL;
    x05 : REAL;
    x06 : REAL;
    x1 : REAL := 1.0;
    x2 : REAL := 2.0;
    x3 : INT := 1;
    x4 : INT := 2;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.5: *FBD: FBD_OP_ARITH*

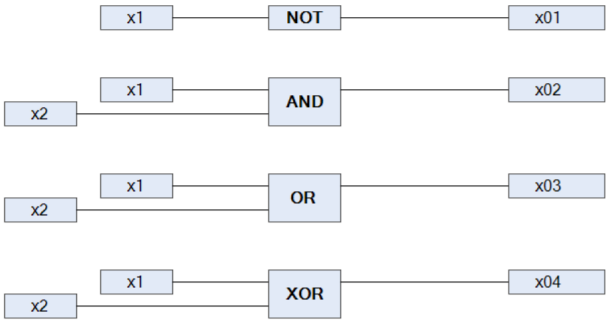
```
FUNCTION_BLOCK FBD_BENDING
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
    OUT3: BOOL;
  END_VAR
  VAR
    PT5s: TIME;
    PT05s: TIME;
    PT1s: TIME;
    TON0: TON;
    TON1: TON;
    TON2: TON;
    TON3: TON;
    TOF0: TON;
    RS0: RS;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.6: *FBD: FBD_BENDING*

```
FUNCTION_BLOCK FBD_OP_BOOL
  VAR
    x01 : BOOL;
    x02 : BOOL;
    x03 : BOOL;
    x04 : BOOL;
    x1 : BOOL := TRUE;
    x2 : BOOL := FALSE;
  END_VAR
```

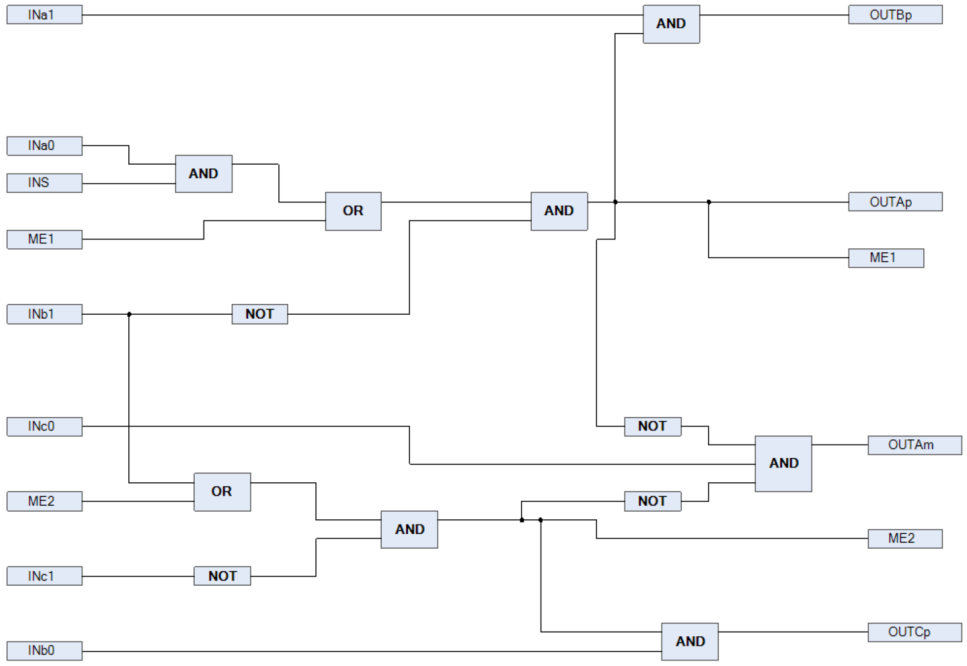



```
END_FUNCTION_BLOCK
```

Listing E.7: *FBD: FBD_OP_BOOL*

```
FUNCTION_BLOCK FBD_CYLINDER
  VAR_INPUT
    INa1: BOOL;
    INa0: BOOL;
    INS: BOOL;
    INb1: BOOL;
    INc0: BOOL;
    INc1: BOOL;
    INb0: BOOL;
  END_VAR
  VAR_OUTPUT
```

```
12      OUTBp: BOOL;
13      OUTAp: BOOL;
14      OUTAm: BOOL;
15      OUTCp: BOOL;
16    END_VAR
17    VAR
18      ME1: BOOL;
19      ME2: BOOL;
20    END_VAR
```




```
22  END_FUNCTION_BLOCK
```

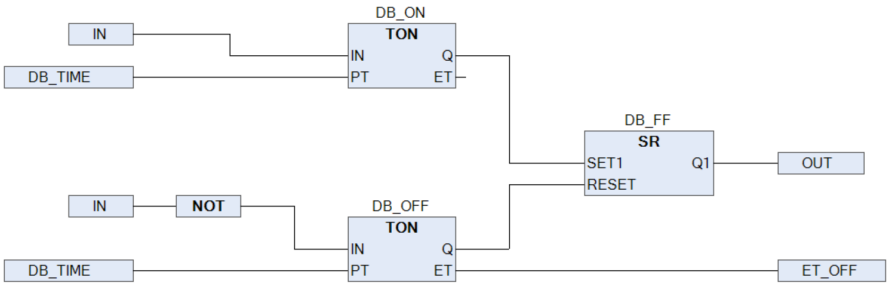
Listing E.8: *FBD: FBD_CYLINDER*

```
1   FUNCTION_BLOCK FBD_DATATYPES
2     VAR
3       A1 : BOOL;
4       A2 : BOOL := TRUE;
5       A3 : BYTE;
6       A4 : WORD;
7       A5 : INT;
8       A6 : INT := 2;
9       A7 : DINT;
10      A8 : DINT := 2;
11      A9 : UINT;
12      A10 : UINT := 2;
13      A11 : UDINT;
14      A12 : UDINT := 2;
15      A13 : REAL;
16      A14 : REAL := 1.23;
17      A15 : TIME;
18      A16 : TIME := T#5000MS;
19      A17 : ARRAY [0..2] OF BOOL;
20      A18 : ARRAY [0..2] OF BYTE;
21      A19 : ARRAY [0..2] OF WORD;
22      A20 : ARRAY [0..2] OF INT;
23      A21 : ARRAY [0..2] OF DINT;
24      A22 : ARRAY [0..2] OF UINT;
```

```
    A23 : ARRAY [0..2] OF UDINT;
    A24 : ARRAY [0..2] OF REAL;
    A25 : ARRAY [0..2] OF TIME;
  END_VAR
END_FUNCTION_BLOCK
```

Listing E.9: *FBD: FBD_DATATYPES*

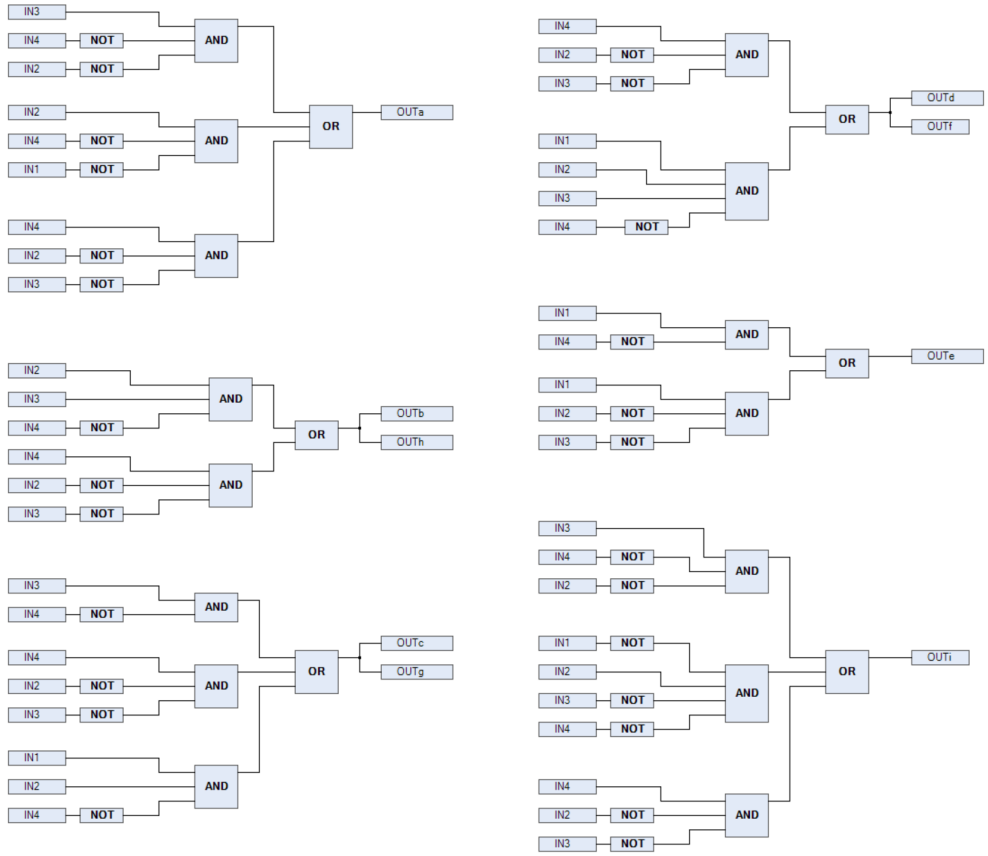
```
FUNCTION_BLOCK FBD_DEBOUNCE
  VAR_INPUT
    IN: BOOL;
    DB_TIME: TIME;
  END_VAR
  VAR_OUTPUT
    OUT: BOOL;
    ET_OFF: TIME;
  END_VAR
  VAR
    DB_ON: TON;
    DB_OFF: TON;
    DB_FF: SR;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.10: *FBD: FBD_DEBOUNCE*

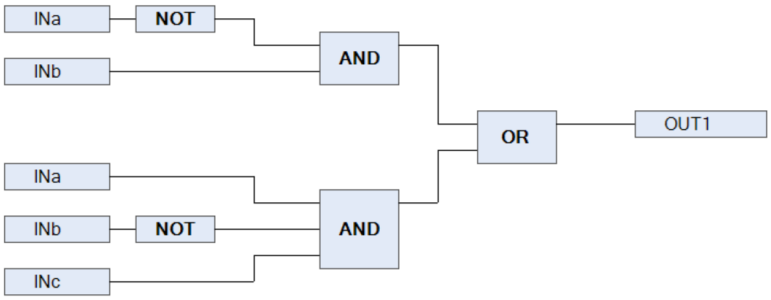
```
FUNCTION_BLOCK FBD_DICE
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUTa: BOOL;
    OUTb: BOOL;
    OUTc: BOOL;
    OUTd: BOOL;
    OUTe: BOOL;
    OUTf: BOOL;
    OUTg: BOOL;
    OUTh: BOOL;
    OUTi: BOOL;
  END_VAR
```

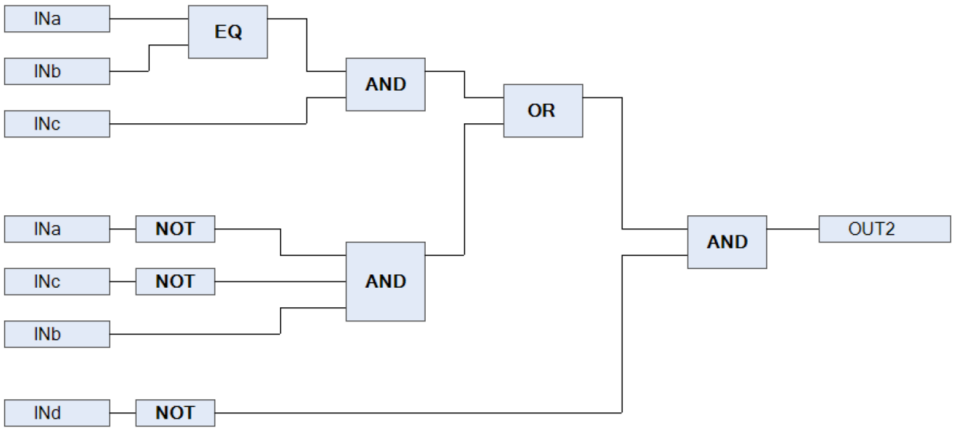



```
END_FUNCTION_BLOCK
```

Listing E.11: *FBD: FBD_DICE*

```
FUNCTION_BLOCK FBD_KV_DIAG
  VAR_INPUT
    INa: BOOL;
    INb: BOOL;
    INc: BOOL;
    INd: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
  END_VAR
```

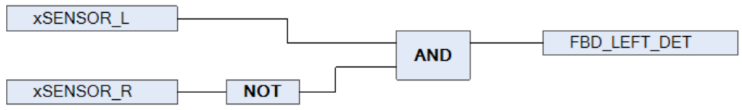





```
END_FUNCTION_BLOCK
```

Listing E.12: *FBD: FBD_KV_DIAG*

```
FUNCTION FBD_LEFT_DET: BOOL
  VAR_INPUT
    xSENSOR_L : BOOL;
    xSENSOR_R : BOOL;
  END_VAR
```

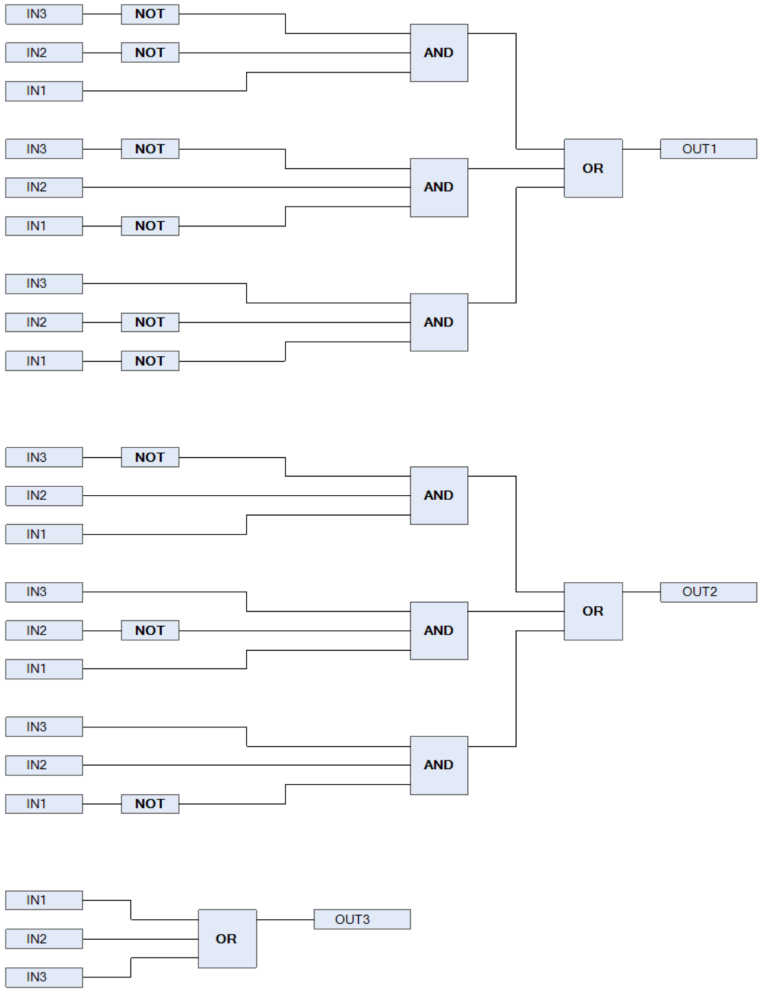



```
END_FUNCTION
```

Listing E.13: *FBD: FBD_LEFT_DET*

```
FUNCTION_BLOCK FBD_POLL
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
```

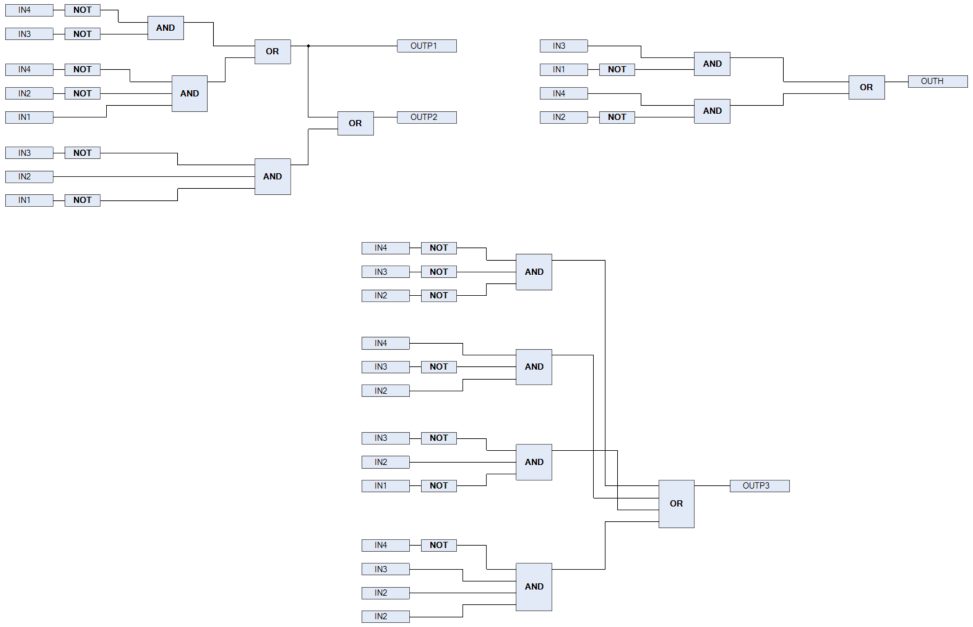
```
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
    OUT3: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.14: *FBD: FBD_POLL*

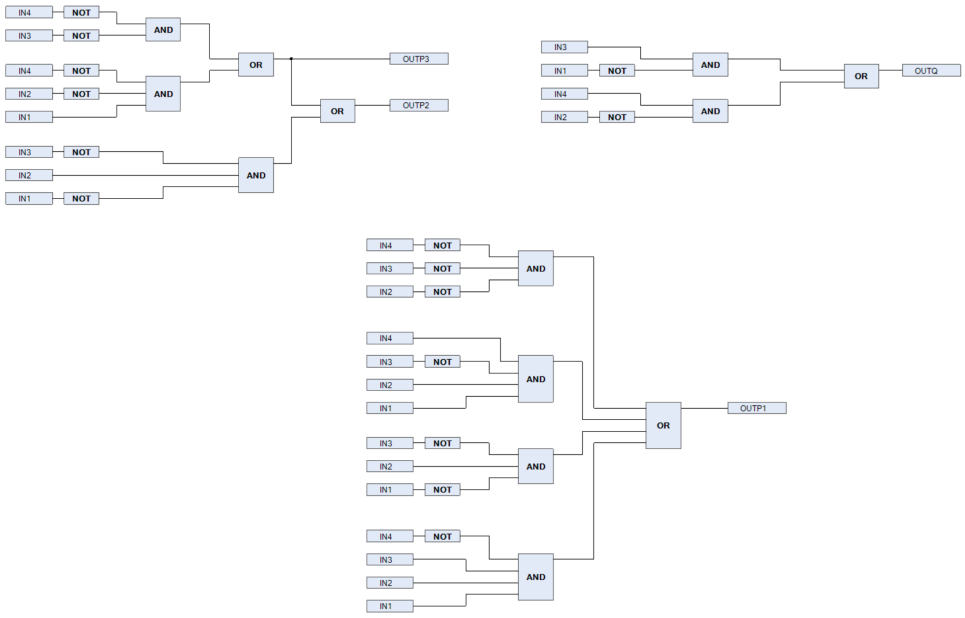
```
FUNCTION_BLOCK FBD_RES_CTRL1
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUTP1: BOOL;
    OUTP2: BOOL;
    OUTP3: BOOL;
    OUTH: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.15: *FBD: FBD_RES_CTRL1*

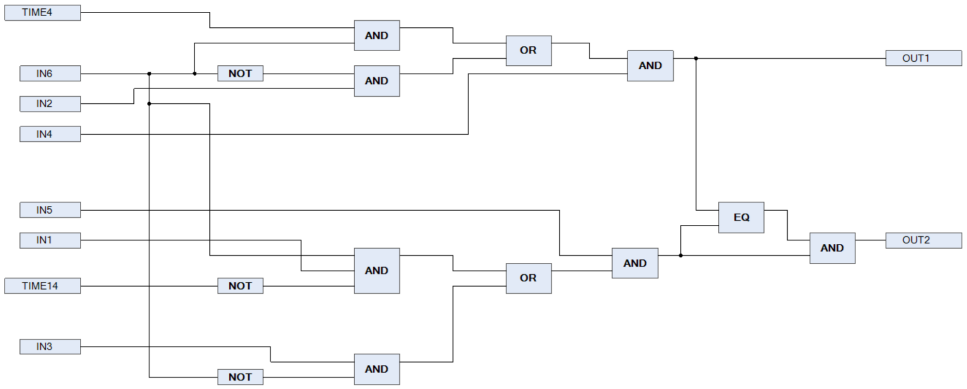
```
FUNCTION_BLOCK FBD_RES_CTRL2
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUTP1: BOOL;
    OUTP2: BOOL;
    OUTP3: BOOL;
    OUTQ: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.16: *FBD: FBD_RES_CTRL2*

```
FUNCTION_BLOCK FBD_ROLL_DOWN
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
    IN5: BOOL;
    IN6: BOOL;
    TIME14: BOOL;
    TIME4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
  END_VAR
```

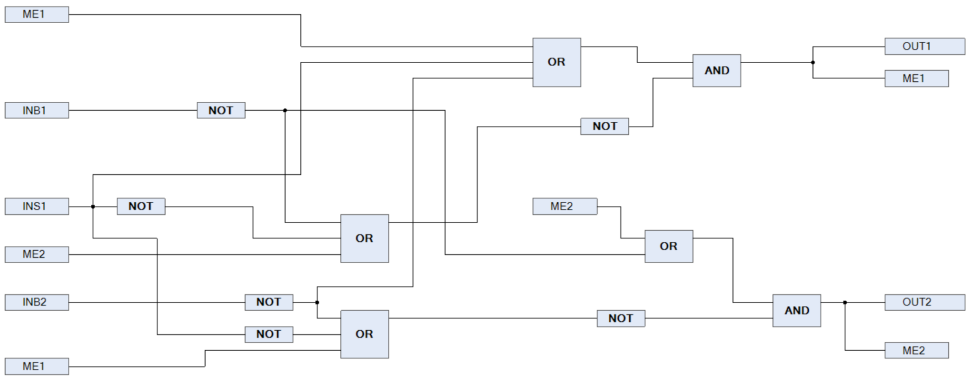



```
END_FUNCTION_BLOCK
```

Listing E.17: *FBD: FBD_ROLL_DOWN*

```
FUNCTION_BLOCK FBD_CABLE_WINCH
```

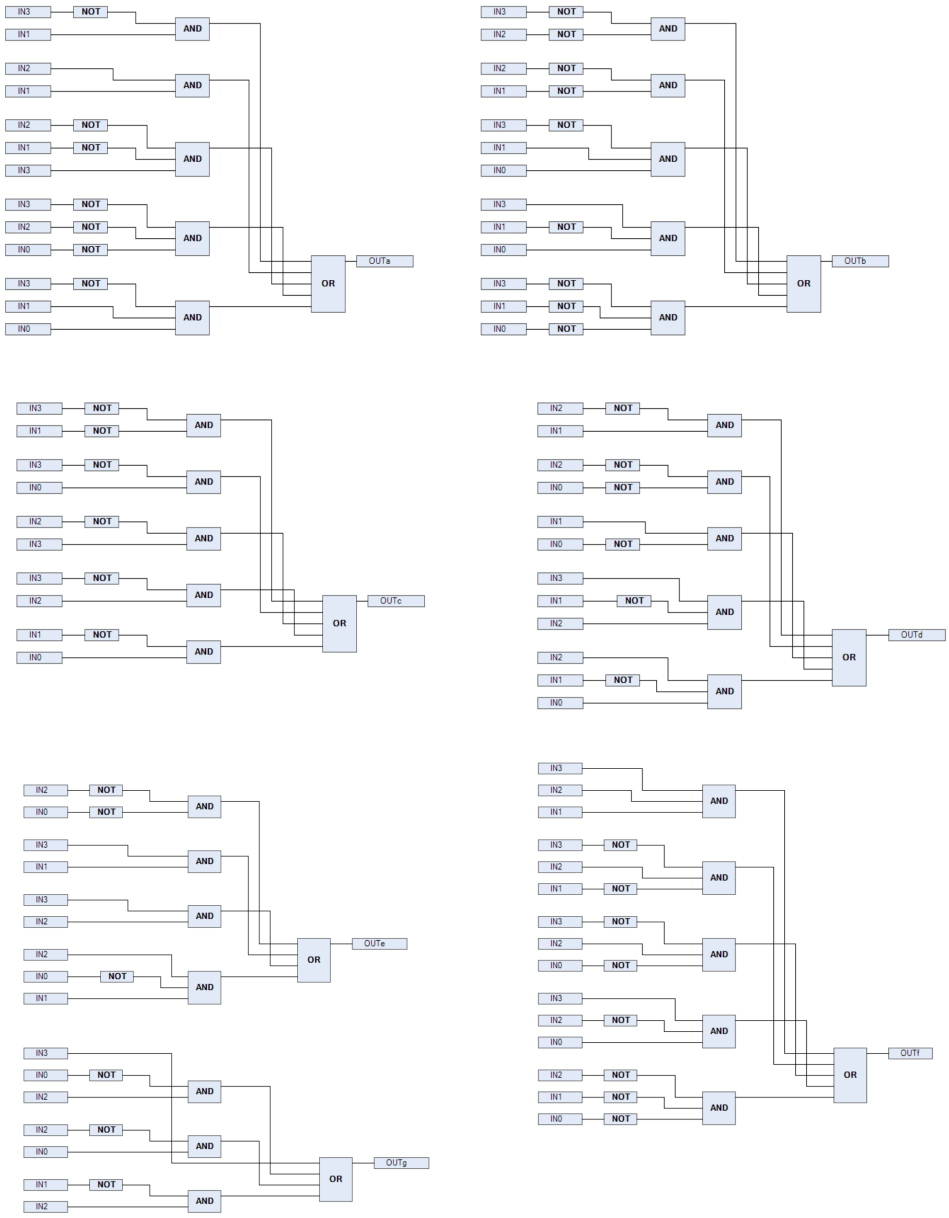
```
  VAR_INPUT
    INS1: BOOL;
    INB1: BOOL;
    INB2: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
  END_VAR
  VAR
    ME1: BOOL;
    ME2: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.18: *FBD: `FBD_CABLE_WINCH`*

```
FUNCTION_BLOCK FBD_SEVEN_SEG
  VAR_INPUT
    IN0: BOOL;
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUTa: BOOL;
    OUTb: BOOL;
    OUTc: BOOL;
    OUTd: BOOL;
    OUTe: BOOL;
    OUTf: BOOL;
    OUTg: BOOL;
  END_VAR
```

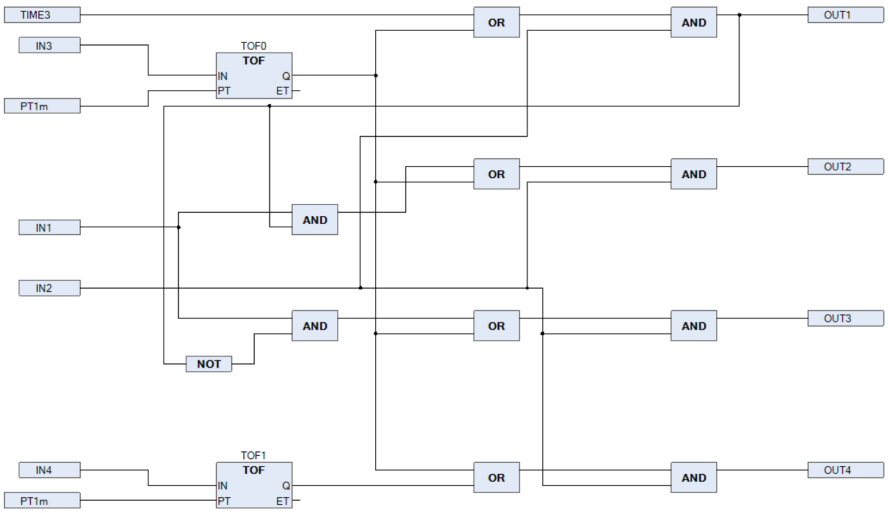



```
END_FUNCTION_BLOCK
```

Listing E.19: *FBD: FBD_SEVEN_SEG*

```
FUNCTION_BLOCK FBD_SHOP_WINDOW
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
    TIME3: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
    OUT3: BOOL;
    OUT4: BOOL;
```

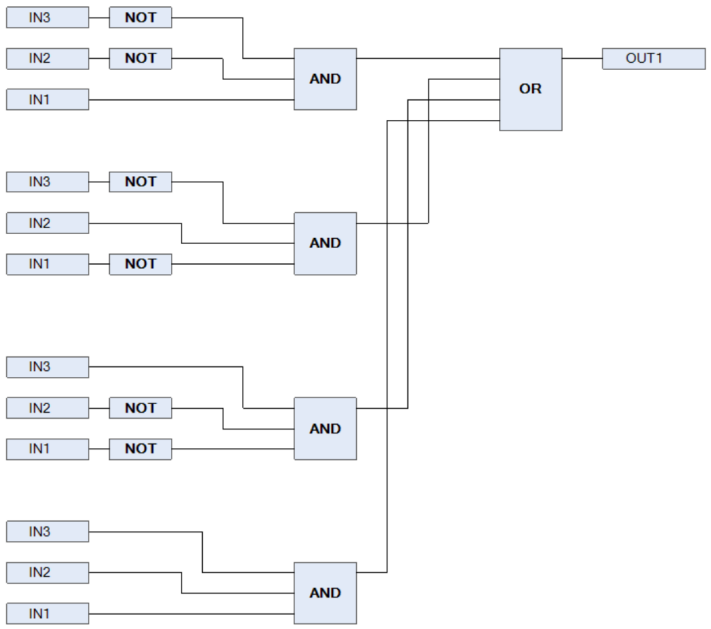
```
14    END_VAR
15    VAR
16      TOF0: TOF;
17      TOF1: TOF;
18      PT1m: TIME;
19    END_VAR
```




```
20
21  END_FUNCTION_BLOCK
```

Listing E.20: *FBD: `FBD_SHOP_WINDOW`*

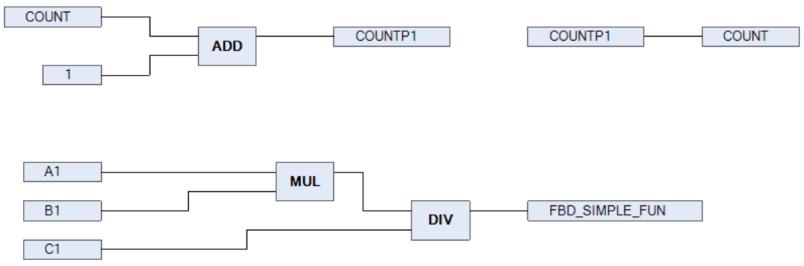
```
1   FUNCTION_BLOCK FBD_SILO_VALVE
2     VAR_INPUT
3       IN1: BOOL;
4       IN2: BOOL;
5       IN3: BOOL;
6       IN4: BOOL;
7     END_VAR
8     VAR_OUTPUT
9       OUT1: BOOL;
10    END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.21: *FBD: FBD_SILO_VALVE*

```
FUNCTION FBD_SIMPLE_FUN: REAL
  VAR_INPUT
    A1: REAL;
    B1: REAL;
    C1: REAL := 1.0;
  END_VAR
  VAR_IN_OUT
    COUNT: INT;
  END_VAR
  VAR
    COUNTP1: INT;
  END_VAR
```

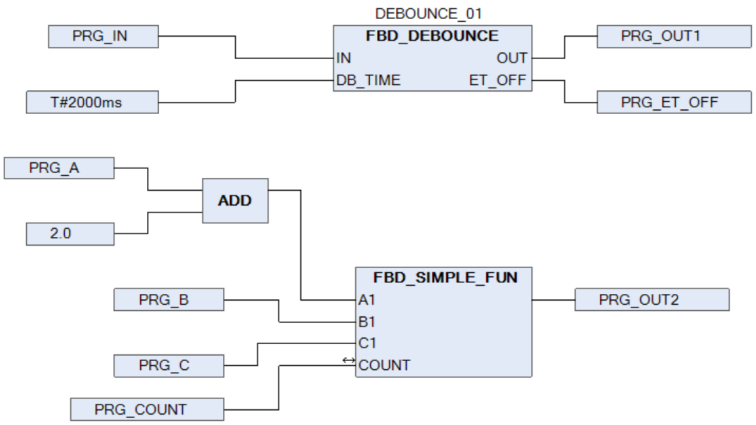



```
END_FUNCTION
```

Listing E.22: *FBD: FBD_SIMPLE_FUN*

```
PROGRAM FBD_SIMPLE_PRG1
  VAR_INPUT
    PRG_IN : BOOL;
    PRG_A : REAL;
    PRG_B : REAL;
    PRG_C : REAL;
```

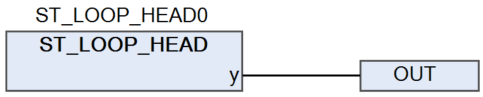
```
7   END_VAR
8   VAR_OUTPUT
9     PRG_OUT1: BOOL;
10    PRG_OUT2: REAL;
11    PRG_ET_OFF: TIME;
12  END_VAR
13  VAR
14    PRG_COUNT: INT := 4;
15    DEBOUNCE_01: FBD_DEBOUNCE;
16  END_VAR
```




```
17
18  END_PROGRAM
```

Listing E.23: *FBD: FBD_SIMPLE_PRG1*

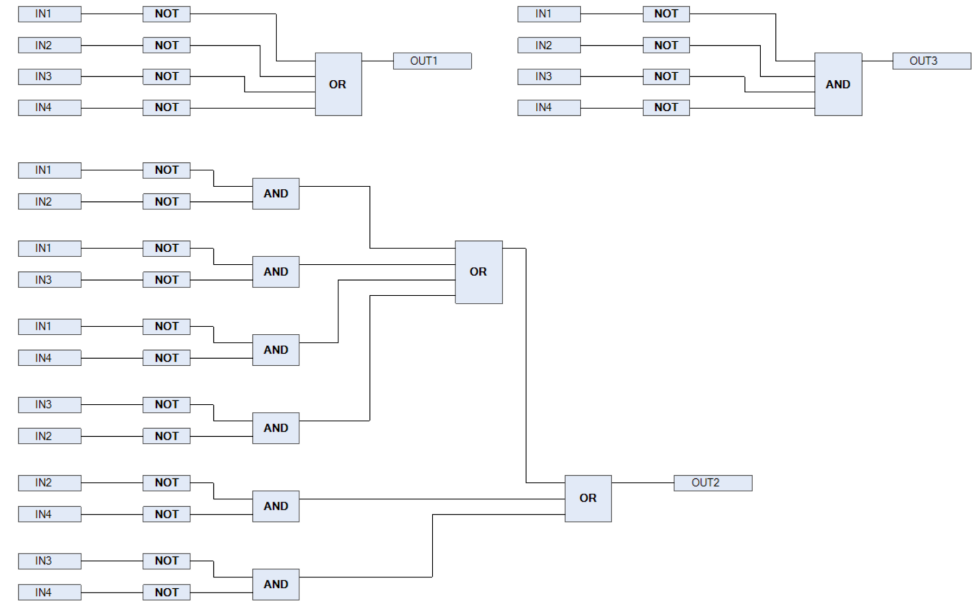
```
1   PROGRAM FBD_SIMPLE_PRG2
2     VAR
3       OUT: REAL;
4       ST_LOOP_HEAD0: ST_LOOP_HEAD;
5     END_VAR
6
```




```
7   END_PROGRAM
```

Listing E.24: *FBD: FBD_SIMPLE_PRG2*

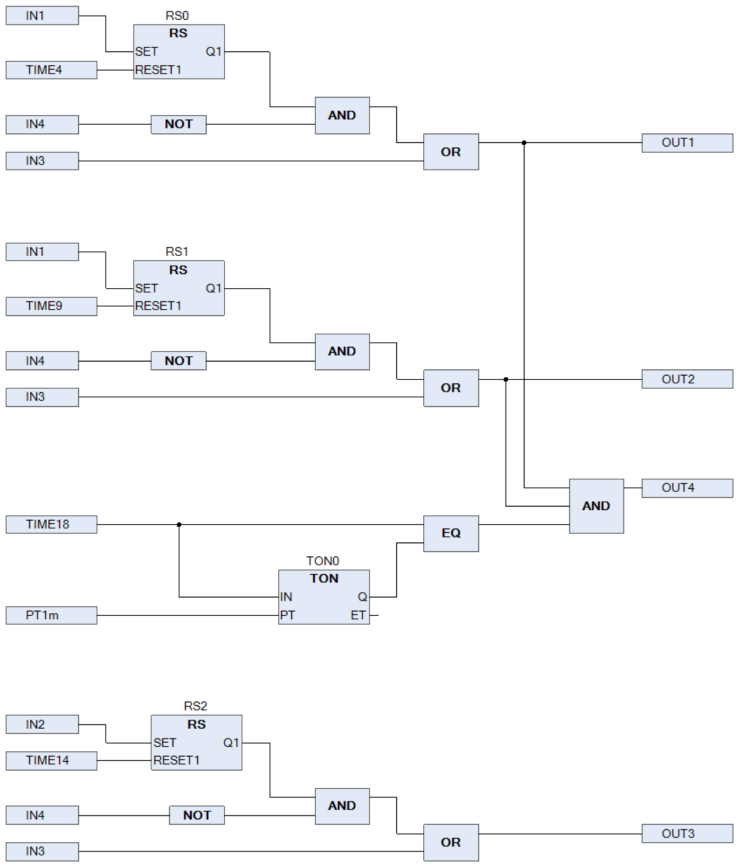
```
1   FUNCTION_BLOCK FBD_SMOKE_DET
2     VAR_INPUT
3       IN1: BOOL;
4       IN2: BOOL;
5       IN3: BOOL;
6       IN4: BOOL;
7     END_VAR
8     VAR_OUTPUT
9       OUT1: BOOL;
10      OUT2: BOOL;
11      OUT3: BOOL;
12    END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.25: *FBD: FBD_SMOKE_DET*

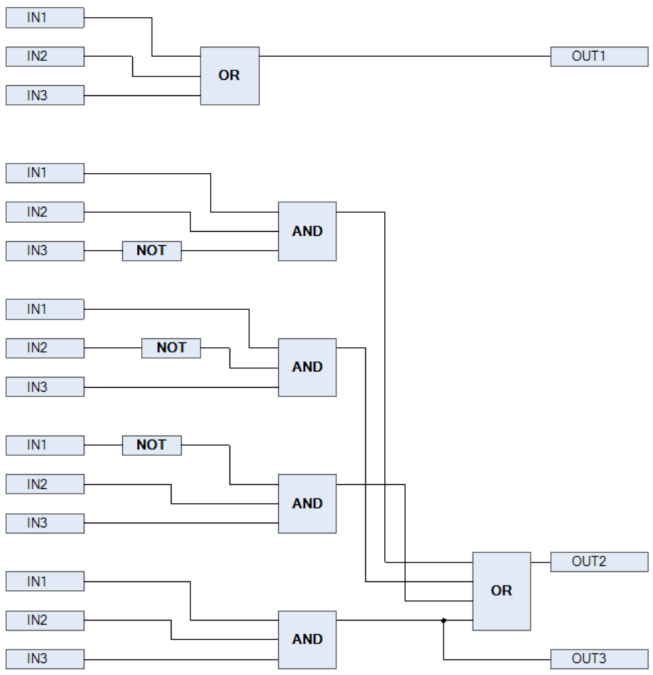
```
FUNCTION_BLOCK FBD_SPORTS_HALL
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
    TIME4: BOOL;
    TIME9: BOOL;
    TIME14: BOOL;
    TIME18: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
    OUT3: BOOL;
    OUT4: BOOL;
  END_VAR
  VAR
    PT1m: TIME;
    RS0: RS;
    RS1: RS;
    RS2: RS;
    TON0: TON;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.26: *FBD: FBD_SPORTS_HALL*

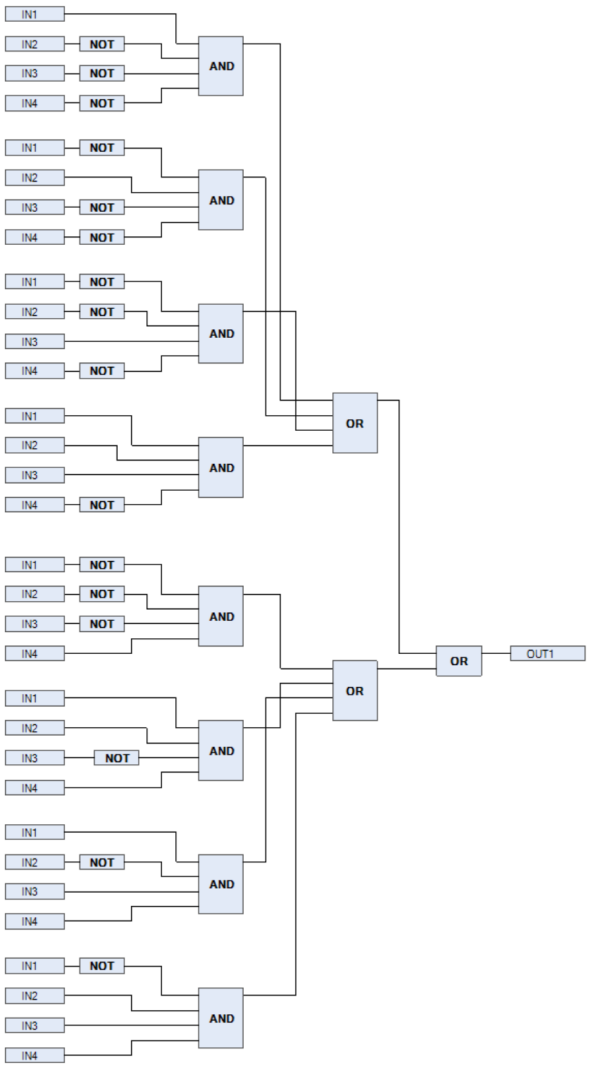
```
FUNCTION_BLOCK FBD_THER_CODE
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
    OUT3: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.27: *FBD: FBD_THER_CODE*

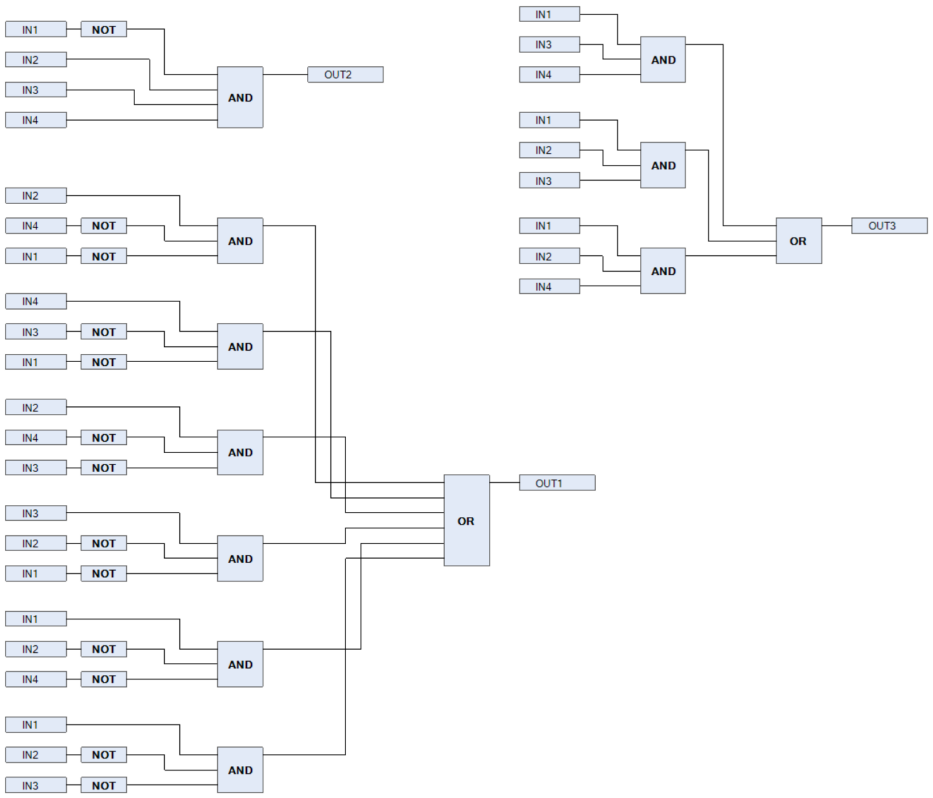
```
FUNCTION_BLOCK FBD_TOGGLE_SWITCH
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.28: *FBD: FBD_TOGGLE_SWITCH*

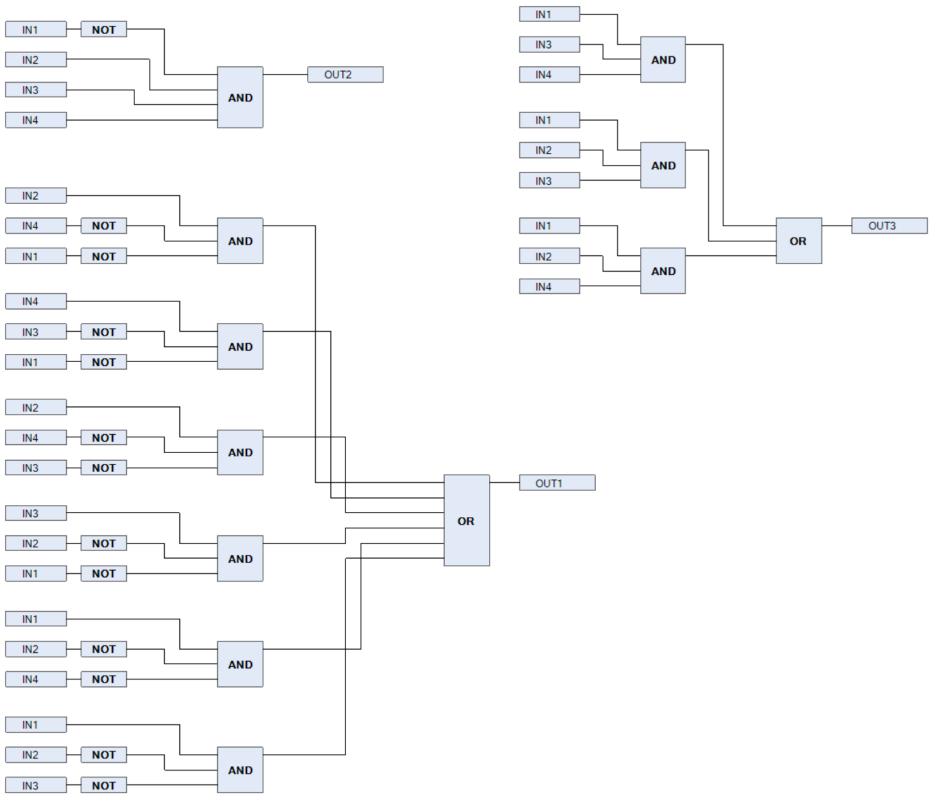
```
FUNCTION_BLOCK FBD_VENT_CTRL
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
    IN4: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
    OUT3: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.29: *FBD: FBD_VENT_CTRL*

```
FUNCTION_BLOCK FBD_WIND_DIR
  VAR_INPUT
    IN1: BOOL;
    IN2: BOOL;
    IN3: BOOL;
  END_VAR
  VAR_OUTPUT
    OUT1: BOOL;
    OUT2: BOOL;
    OUT3: BOOL;
    OUT4: BOOL;
  END_VAR
```




```
END_FUNCTION_BLOCK
```

Listing E.30: *FBD: `FBD_WIND_DIR`*

Appendix F

Resulting FBD-Based Quartz Models

```
module FBD_TWO_OF_THREE(
    event bool ?EI,
    event bool !EO,
    bool ?xB1_Temp,
    bool ?xB2_Temp,
    bool ?xB3_Temp,
    bool !xP1_Temp){

    loop{
        immediate await(EI);

        xP1_Temp = ((xB1_Temp&xB2_Temp)|(xB1_Temp&xB3_Temp)|(xB2_Temp&
            xB3_Temp));

        emit(EO); pause;
    }
}
```

Listing F.1: *Quartz Model: FBD_TWO_OF_THREE*

```
import RS.*;

module FBD_AIR_COND_CTRL(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool ?IN5,
    bool ?IN6,
    bool ?IN7,
    bool ?IN8,
    bool ?IN9,
    bool !OUT1,
    bool !OUT2){

    event bool RS0_EI;
    event bool RS0_EO;
    bool RS0_Q1;

    loop{
```

```
23          immediate await(EI);
24
25          emit(RS0_EI);
26          immediate await(RS0_EO);
27
28          OUT2 = RS0_Q1;
29          emit(EO); pause;
30      }
31      || RS0:RS(RS0_EI, RS0_EO,
32          (IN1|IN2|IN3|IN4),
33          ((!(IN5)|!(IN6)|!(IN7)|!(IN8))|!(IN9)),
34          RS0_Q1);
35  }
```

Listing F.2: *Quartz Model: FBD_AIR_COND_CTRL*

```
1   module FBD_ALARM(
2       bool ?xSENSOR_L,
3       bool ?xSENSOR_M,
4       bool ?xSENSOR_R,
5       bool !FBD_ALARM){
6
7       FBD_ALARM = (!(xSENSOR_L)&!(xSENSOR_M)&!(xSENSOR_R))|(xSENSOR_L&
            xSENSOR_R);
8   }
```

Listing F.3: *Quartz Model: FBD_ALARM*

```
1   module FBD_ANTIVALENCE(
2       event bool ?EI,
3       event bool !EO,
4       bool ?IN0,
5       bool ?IN1,
6       bool ?IN2,
7       bool !OUT1){
8
9       loop{
10          immediate await(EI);
11          OUT1 = ((!(IN0)&!(IN1)&IN2)|(!(IN0)&!(IN2)&IN1)|(!(IN1)&!(IN2)&
                IN0));
12          emit(EO); pause;
13      }
14  }
```

Listing F.4: *Quartz Model: FBD_ANTIVALENCE*

```
1   module FBD_OP_ARITH(
2       event bool ?EI,
3       event bool !EO){
4
5       real x01;
6       real x02;
7       real x03;
8       real x04;
9       real x05;
10      real x06;
11      real x1;
12      real x2;
13      int{32768} x3;
14      int{32768} x4;
15
16      x1 = 1.0;
17      x2 = 2.0;
18      x3 = 1;
19      x4 = 2;
```

```
    pause;

    loop{
        immediate await(EI);

        x01 = x1 + x2;
        x02 = x1 - x2;
        x03 = x1 * x2;
        x04 = x1 / x2;
        x05 = exp(x1, x2);
        x06 = x3 % x4;

        emit(EO); pause;
    }
}
```

Listing F.5: *Quartz Model:* `FBD_OP_ARITH`

```
import TON.*;
import TOF.*;
import RS.*;

module FBD_BENDING(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool !OUT1,
    bool !OUT2,
    bool !OUT3){

    nat PT5s;
    nat PT05s;
    nat PT1s;

    event bool TON0_EI;
    event bool TON0_EO;
    event bool TON1_EI;
    event bool TON1_EO;
    event bool TON2_EI;
    event bool TON2_EO;
    event bool TOF0_EI;
    event bool TOF0_EO;
    event bool TON3_EI;
    event bool TON3_EO;
    event bool RS0_EI;
    event bool RS0_EO;

    bool TON0_Q;
    nat TON0_ET;
    bool TON1_Q;
    nat TON1_ET;
    bool TON2_Q;
    nat TON2_ET;
    bool TOF0_Q;
    nat TOF0_ET;
    bool TON3_Q;
    nat TON3_ET;
    bool RS0_Q1;


    loop{
        immediate await(EI);

```

```
49          emit(TON0_EI);
50          immediate await(TON0_EO);
51
52          emit(TON1_EI);
53          immediate await(TON1_EO);
54
55          emit(TON2_EI);
56          immediate await(TON2_EO);
57
58          emit(TOF0_EI);
59          immediate await(TOF0_EO);
60
61          OUT1 = (IN1&TON0_Q)&!((TON1_Q&TON2_Q))&!((((!(TON2_Q)&TOF0_Q)&IN3
                ));
62          OUT2 = (!((((!(TON2_Q)&TOF0_Q)&IN3))&(!(TON2_Q)&TOF0_Q));
63
64          emit(TON3_EI);
65          immediate await(TON3_EO);
66
67          emit(RS0_EI);
68          immediate await(RS0_EO);
69
70          OUT3 = RS0_Q1;
71
72          emit(EO); pause;
73      }
74      || TON0:TON(TON0_EI, TON0_EO, CLK,
75          IN1,
76          PT1s,
77          TON0_Q, TON0_ET);
78      || TON1:TON(TON1_EI, TON1_EO, CLK,
79          (IN1&TON0_Q),
80          PT1s,
81          TON1_Q, TON1_ET);
82      || TON2:TON(TON2_EI, TON2_EO, CLK,
83          IN2,
84          PT05s,
85          TON2_Q, TON2_ET);
86      || TOF0:TOF(TOF0_EI, TOF0_EO, CLK,
87          (IN1&TON0_Q),
88          PT1s,
89          TOF0_Q, TOF0_ET);
90      || TON3:TON(TON3_EI, TON3_EO, CLK,
91          !(IN1),
92          PT5s,
93          TON3_Q, TON3_ET);
94      || RS0:RS(RS0_EI, RS0_EO,
95          ((IN1&TON0_Q)|TON3_Q),
96          (TON1_Q&TON2_Q),
97          RS0_Q1);
98  }
```

Listing F.6: *Quartz Model:* `FBD_BENDING`

```
1   module FBD_OP_BOOL(
2       event bool ?EI,
3       event bool !EO){
4
5       bool x01;
6       bool x02;
7       bool x03;
8       bool x04;
9       bool x1;
10      bool x2;
11
12      x1 = true;
```

```
13      x2 = false;
14      pause;
15
16      loop{
17          immediate await(EI);
18
19          x01 = !(x1);
20          x02 = x1 & x2;
21          x03 = x1 | x2;
22          x04 = x1 ^ x2;
23
24          emit(EO); pause;
25      }
26  }
```

Listing F.7: *Quartz Model: FBD_OP_BOOL*

```
1   module FBD_CYLINDER(
2       event bool ?EI,
3       event bool !EO,
4       bool ?INa1,
5       bool ?INa0,
6       bool ?INS,
7       bool ?INb1,
8       bool ?INc0,
9       bool ?INc1,
10      bool ?INb0,
11      bool !OUTBp,
12      bool !OUTAp,
13      bool !OUTAm,
14      bool !OUTCp){
15
16      bool ME1;
17      bool ME2;
18
19      loop{
20          immediate await(EI);
21
22          OUTAp = ((ME1|(INa0&INS))&!(INb1));
23          pause;
24          ME1 = ((ME1|(INa0&INS))&!(INb1));
25          OUTBp = (INa1&((ME1|(INa0&INS))&!(INb1)));
26          ME2 = ((ME2|INb1)&!(INc1));
27          OUTAm = (!(((ME1|(INa0&INS))&!(INb1)))&INc0&!(((ME2|INb1)&!(INc1
                ))));
28          OUTCp = (((ME2|INb1)&!(INc1))&INb0);
29
30          emit(EO); pause;
31      }
32  }
```

Listing F.8: *Quartz Model: FBD_CYLINDER*

```
1   module FBD_DATATYPES(
2       event bool ?EI,
3       event bool !EO){
4
5       bool A1;
6       bool A2;
7       bv{16} A3;
8       bv{32} A4;
9       int{32768} A5;
10      int{32768} A6;
11      int{2147483648} A7;
12      int{2147483648} A8;
```

```
    nat{65536} A9;
    nat{65536} A10;
    nat{4294967296} A11;
    nat{4294967296} A12;
    real A13;
    real A14;
    nat A15;
    nat A16;
    [3]bool A17;
    [3]bv{16} A18;
    [3]bv{32} A19;
    [3]int{32768} A20;
    [3]int{2147483648} A21;
    [3]nat{65536} A22;
    [3]nat{4294967296} A23;
    [3]real A24;
    [3]nat A25;

    A2 = true;
    A6 = 2;
    A8 = 2;
    A10 = 2;
    A12 = 2;
    A14 = 1.23;
    A16 = 5000;
    pause;

    loop{
        immediate await(EI);
        emit(EO); pause;
    }
}
```

Listing F.9: *Quartz Model:* `FBD_DATATYPES`

```
import TON.*;
import SR.*;

module FBD_DEBOUNCE(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
    bool ?IN,
    nat ?DB_TIME,
    bool !OUT,
    nat    !ET_OFF){

    event bool DB_ON_EI;
    event bool DB_ON_EO;
    event bool DB_OFF_EI;
    event bool DB_OFF_EO;
    event bool DB_FF_EI;
    event bool DB_FF_EO;

    bool DB_ON_Q;
    bool DB_OFF_Q;
    bool DB_FF_Q1;
    nat DB_ON_ET;
    nat DB_OFF_ET;

    loop{
        immediate await(EI);

        emit(DB_ON_EI);
        immediate await(DB_ON_EO);

```

```
32          emit(DB_OFF_EI);
33          immediate await(DB_OFF_EO);
34
35          ET_OFF = DB_OFF_ET;
36
37          emit(DB_FF_EI);
38          immediate await(DB_FF_EO);
39
40          OUT = DB_FF_Q1;
41
42          emit(EO);
43          pause;
44      }
45      || DB_ON:TON(DB_ON_EI, DB_ON_EO, CLK,
46          IN, DB_TIME,
47          DB_ON_Q, DB_ON_ET);
48      || DB_OFF:TON(DB_OFF_EI, DB_OFF_EO, CLK,
49          !(IN),
50          DB_TIME,
51          DB_OFF_Q, DB_OFF_ET);
52      || DB_FF:SR(DB_FF_EI, DB_FF_EO,
53          DB_ON_Q,
54          DB_OFF_Q, DB_FF_Q1);
55  }
```

Listing F.10: *Quartz Model: FBD_DEBOUNCE*

```
1   module FBD_DICE(
2       event bool ?EI,
3       event bool !EO,
4       bool ?IN1,
5       bool ?IN2,
6       bool ?IN3,
7       bool ?IN4,
8       bool !OUTa,
9       bool !OUTb,
10      bool !OUTc,
11      bool !OUTd,
12      bool !OUTe,
13      bool !OUTf,
14      bool !OUTg,
15      bool !OUTh,
16      bool !OUTi){
17
18      loop{
19          immediate await(EI);
20
21          OUTa = ((IN3&!(IN4)&!(IN2))|(IN2&!(IN4)&!(IN1))|(IN4&!(IN2)&!(
                IN3)));
22          OUTb = ((IN2&IN3&!(IN4))|(IN4&!(IN2)&!(IN3)));
23          OUTc = ((IN3&!(IN4))|(IN4&!(IN2)&!(IN3))|(IN1&IN2&!(IN4)));
24          OUTd = ((IN4&!(IN2)&!(IN3))|(IN1&IN2&IN3&!(IN4)));
25          OUTe = ((IN1&!(IN4))|(IN1&!(IN3)&!(IN2)));
26          OUTf = ((IN4&!(IN2)&!(IN3))|(IN1&IN2&IN3&!(IN4)));
27          OUTh = ((IN2&IN3&!(IN4))|(IN4&!(IN2)&!(IN3)));
28          OUTg = ((IN3&!(IN4))|(IN4&!(IN2)&!(IN3))|(IN1&IN2&!(IN4)));
29          OUTi = ((IN3&!(IN4)&!(IN2))|(IN2&!(IN4)&!(IN1)&!(IN3))|(IN4&!(
                IN2)&!(IN3)));
30
31          emit(EO); pause;
32      }
33  }
```

Listing F.11: *Quartz Model: FBD_DICE*

```
module FBD_KV_DIAG(
    event bool ?EI,
    event bool !EO,
    bool ?INa,
    bool ?INb,
    bool ?INc,
    bool ?INd,
    bool !OUT1,
    bool !OUT2){

    loop{
        immediate await(EI);

        OUT1 = ((!(INa)&INb)|(INa&!(INb)&INc));
        OUT2 = (((INa==INb&INc)|(!(INa)&!(INc)&INb))&!(INd));

        emit(EO); pause;
    }
}
```

Listing F.12: *Quartz Model: FBD_KV_DIAG*

```
module FBD_LEFT_DET(
    bool ?xSENSOR_L,
    bool ?xSENSOR_R,
    bool !FBD_LEFT_DET){

    FBD_LEFT_DET = (xSENSOR_L & !(xSENSOR_R));
}
```

Listing F.13: *Quartz Model: FBD_LEFT_DET*

```
module FBD_POLL(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool !OUT1,
    bool !OUT2,
    bool !OUT3){

    loop{
        immediate await(EI);

        OUT1 = ((!(IN3)&!(IN2)&IN1)|(!(IN3)&IN2&!(IN1))|(IN3&!(IN2)&!(
            IN1)));
        OUT2 = ((!(IN3)&IN2&IN1)|(IN3&!(IN2)&IN1)|(IN3&IN2&!(IN1)));
        OUT3 = (IN1&IN2&IN3);

        emit(EO); pause;
    }
}
```

Listing F.14: *Quartz Model: FBD_POLL*

```
module FBD_RES_CTRL1(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool !OUTP1,
    bool !OUTP2,
```

```
    bool !OUTP3,
    bool !OUTH){

    loop{
        immediate await(EI);

        OUTP1 = ((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1));
        OUTP2 = (((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1))|(!(IN3)&IN2&!(IN1
            )));
        OUTP3 = ((!(IN4)&!(IN3)&!(IN2))|(IN4&!(IN3)&IN2)|(!(IN3)&IN2&!(
            IN1))|(!(IN4)&IN3&IN2&IN2));
        OUTH = ((IN3&!(IN1))|(IN4&!(IN2)));

        emit(EO); pause;
    }
}
```

Listing F.15: *Quartz Model: FBD_RES_CTRL1*

```
module FBD_RES_CTRL2(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool !OUTP1,
    bool !OUTP2,
    bool !OUTP3,
    bool !OUTQ){

    loop{
        immediate await(EI);

        OUTP3 = ((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1));
        OUTP2 = (((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1))|(!(IN3)&IN2&!(IN1
            )));
        OUTP1 = ((!(IN4)&!(IN3)&!(IN2))|(IN4&!(IN3)&IN2&IN1)|(!(IN3)&IN2
            &!(IN1))|(!(IN4)&IN3&IN2&IN1));
        OUTQ = ((IN3&!(IN1))|(IN4&!(IN2)));

        emit(EO); pause;
    }
}
```

Listing F.16: *Quartz Model: FBD_RES_CTRL2*

```
module FBD_ROLL_DOWN(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool ?IN5,
    bool ?IN6,
    bool ?TIME14,
    bool ?TIME4,
    bool !OUT1,
    bool !OUT2){

    loop{
        immediate await(EI);

        OUT1 = (((TIME4&IN6)|(!(IN6)&IN2))&IN4);
```

```
        OUT2 = ((((TIME4&IN6)|(!(IN6)&IN2))&IN4)==(IN5&((IN6&IN1&!(
            TIME14))|(IN3&!(IN6))))&(IN5&((IN6&IN1&!(TIME14))|(IN3&!(IN6
            )))));

        emit(EO); pause;
    }
}
```

Listing F.17: *Quartz Model:* `FBD_ROLL_DOWN`

```
module FBD_CABLE_WINCH(
    event bool ?EI,
    event bool !EO,
    bool ?INS1,
    bool ?INB1,
    bool ?INB2,
    bool !OUT1,
    bool !OUT2){

    bool ME1;
    bool ME2;

    loop{
        immediate await(EI);

        OUT1 = ((ME1|INS1|!(INB2))&!((!(INB1)|!(INS1)|ME2)));
        pause;
        ME1 = ((ME1|INS1|!(INB2))&!((!(INB1)|!(INS1)|ME2)));
        OUT2 = ((ME2|!(INB1))&!((!(INB2)|!(INS1)|ME1)));
        pause;
        ME2 = ((ME2|!(INB1))&!((!(INB2)|!(INS1)|ME1)));

        emit(EO); pause;
    }
}
```

Listing F.18: *Quartz Model:* `FBD_CABLE_WINCH`

```
module FBD_SEVEN_SEG(
    event bool ?EI,
    event bool !EO,
    bool ?IN0,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool !OUTa,
    bool !OUTb,
    bool !OUTc,
    bool !OUTd,
    bool !OUTe,
    bool !OUTf,
    bool !OUTg){

    loop{
        immediate await(EI);

        OUTa = ((!(IN3)&IN1)|(IN2&IN1)|(!(IN2)&!(IN1)&IN3)|(!(IN3)&!(IN2
            )&!(IN0))|(!(IN3)&IN1&IN0));
        OUTb = ((!(IN3)&!(IN2))|(!(IN2)&!(IN1))|(!(IN3)&IN1&IN0)|(IN3&!(
            IN1)&IN0)|(!(IN3)&!(IN1)&!(IN0)));
        OUTc = ((!(IN3)&!(IN1))|(!(IN3)&IN0)|(!(IN2)&IN3)|(!(IN3)&IN2)
            |(!(IN1)&IN0));
        OUTd = ((!(IN2)&IN1)|(!(IN2)&!(IN0))|(IN1&!(IN0))|(IN3&!(IN1)&
            IN2)|(IN2&!(IN1)&IN0));
```

```
        OUTe = ((!(IN2)&!(IN0))|(IN3&IN1)|(IN3&IN2)|(IN2&!(IN0)&IN1));
        OUTf = ((IN3&IN2&IN1)|(!(IN3)&IN2&!(IN1))|(!(IN3)&IN2&!(IN0))|(
            IN3&!(IN2)&IN0)|(!(IN2)&!(IN1)&!(IN0)));
        OUTg = (IN3|(!(IN0)&IN2)|(!(IN2)&IN0)|(!(IN1)&IN2));

        emit(EO); pause;
    }
}
```

Listing F.19: *Quartz Model:* `FBD_SEVEN_SEG`

```
import TOF.*;

module FBD_SHOP_WINDOW(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool ?TIME3,
    bool !OUT1,
    bool !OUT2,
    bool !OUT3,
    bool !OUT4){

    nat PT1m;

    event bool TOF0_EI;
    event bool TOF0_EO;
    event bool TOF1_EI;
    event bool TOF1_EO;

    bool TOF0_Q;
    nat TOF0_ET;
    bool TOF1_Q;
    nat TOF1_ET;

    loop{
        immediate await(EI);

        emit(TOF0_EI);
        immediate await(TOF0_EO);

        OUT1 = ((TIME3|TOF0_Q)&IN2);
        OUT2 = (IN2&((((TIME3|TOF0_Q)&IN2)&IN1)|TOF0_Q));
        OUT3 = (IN2&((IN1&!(((TIME3|TOF0_Q)&IN2)))|TOF0_Q));

        emit(TOF1_EI);
        immediate await(TOF1_EO);

        OUT4 = (IN2&(TOF0_Q|TOF1_Q));

        emit(EO); pause;
    }
    || TOF0:TOF(TOF0_EI, TOF0_EO, CLK,
        IN3,
        PT1m,
        TOF0_Q, TOF0_ET);
    || TOF1:TOF(TOF1_EI, TOF1_EO, CLK,
        IN4,
        PT1m,
        TOF1_Q, TOF1_ET);
```

```
}
```

Listing F.20: *Quartz Model:* `FBD_SHOP_WINDOW`

```
module FBD_SILO_VALVE(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool !OUT1){

    loop{
        immediate await(EI);

        OUT1 = ((!(IN3)&!(IN2)&IN1)|(!(IN3)&IN2&!(IN1))|(IN3&!(IN2)&!(
            IN1))|(IN3&IN2&IN1));

        emit(EO); pause;
    }
}
```

Listing F.21: *Quartz Model:* `FBD_SILO_VALVE`

```
module FBD_SIMPLE_FUN(
    real ?A1,
    real ?B1,
    real ?C1,
    int{32768} COUNT,
    real !FBD_SIMPLE_FUN){

    int{32768} COUNTP1;

    COUNTP1 = COUNT+1;
    pause;
    COUNT = COUNTP1;
    FBD_SIMPLE_FUN = (A1*B1)/C1;
}
```

Listing F.22: *Quartz Model:* `FBD_SIMPLE_FUN`

```
import FBD_DEBOUNCE.*;
import FBD_SIMPLE_FUN.*;

module FBD_SIMPLE_PRG1(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
    bool ?PRG_IN,
  real ?PRG_A,
    real ?PRG_B,
    real ?PRG_C,
    bool !PRG_OUT1,
    nat !PRG_ET_OFF,
    real !PRG_OUT2){

    int{32768} PRG_COUNT;

    event bool DEBOUNCE_01_EI;
    event bool DEBOUNCE_01_EO;

    bool DEBOUNCE_01_OUT;
    nat DEBOUNCE_01_ET_OFF;

```

```
24      real FBD_SIMPLE_FUN_l1;
25
26      PRG_COUNT = 4;
27      pause;
28
29      loop{
30          immediate await(EI);
31
32          emit(DEBOUNCE_01_EI);
33          immediate await(DEBOUNCE_01_EO);
34
35          PRG_OUT1 = DEBOUNCE_01_OUT;
36          PRG_ET_OFF = DEBOUNCE_01_ET_OFF;
37
38          FBD_SIMPLE_FUN(
39              PRG_A + 2.0,
40              PRG_B, PRG_C,
41              PRG_COUNT, FBD_SIMPLE_FUN_l1);
42
43          PRG_OUT2 = FBD_SIMPLE_FUN_l1;
44
45          emit(EO);
46          pause;
47      }
48      || DEBOUNCE_01:FBD_DEBOUNCE(DEBOUNCE_01_EI, DEBOUNCE_01_EO, CLK,
49          PRG_IN,
50          2000,
51          DEBOUNCE_01_OUT, DEBOUNCE_01_ET_OFF);
52  }
```

Listing F.23: *Quartz Model: FBD_SIMPLE_PRG1*

```
1   import ST_LOOP_HEAD.*;
2
3   module FBD_SIMPLE_PRG2(
4       event bool ?EI,
5       event bool !EO){
6
7       event bool ST_LOOP_HEAD0_EI;
8       event bool ST_LOOP_HEAD0_EO;
9
10      int{32768} ST_LOOP_HEAD0_y;
11      int{32768} OUT;
12
13      loop{
14          immediate await(EI);
15
16          emit(ST_LOOP_HEAD0_EI);
17          immediate await(ST_LOOP_HEAD0_EO);
18
19          OUT = ST_LOOP_HEAD0_y;
20
21          emit(EO); pause;
22      }
23      || ST_LOOP_HEAD0:ST_LOOP_HEAD(ST_LOOP_HEAD0_EI, ST_LOOP_HEAD0_EO,
24      ST_LOOP_HEAD0_y);
25  }
```

Listing F.24: *Quartz Model: FBD_SIMPLE_PRG2*

```
1   module FBD_SMOKE_DET(
2       event bool ?EI,
3       event bool !EO,
4       bool ?IN1,
5       bool ?IN2,
```

```
    bool ?IN3,
    bool ?IN4,
    bool !OUT1,
    bool !OUT2,
    bool !OUT3){

    loop{
        immediate await(EI);

        OUT1 = (!(IN1)|!(IN2)|!(IN3)|!(IN4));
        OUT2 = ((((!(IN1)&!(IN2))|(!(IN1)&!(IN3)))|(!(IN1)&!(IN4)&(!(IN3)
            &!(IN2))&(!(IN2)&!(IN4))&(!(IN3)&!(IN4))));
        OUT3 = (!(IN1)&!(IN2)&!(IN3)&!(IN4));

        emit(EO); pause;
    }
}
```

Listing F.25: *Quartz Model:* `FBD_SMOKE_DET`

```
import RS.*;
import TON.*;

module FBD_SPORTS_HALL(
    event bool ?EI,
    event bool !EO,
    nat ?CLK,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool ?TIME4,
    bool ?TIME9,
    bool ?TIME14,
    bool ?TIME18,
    bool !OUT1,
    bool !OUT2,
    bool !OUT3,
    bool !OUT4){

    nat PT1m;

    event bool RS0_EI;
    event bool RS0_EO;
    event bool RS1_EI;
    event bool RS1_EO;
    event bool RS2_EI;
    event bool RS2_EO;
    event bool TON0_EI;
    event bool TON0_EO;

    bool RS0_Q1;
    bool RS1_Q1;
    bool RS2_Q1;
    bool TON0_Q;
    nat TON0_ET;

    loop{
        immediate await(EI);

        emit(RS0_EI);
        immediate await(RS0_EO);

        OUT1 = ((RS0_Q1&!(IN4))|IN3);

        emit(RS1_EI);
```

```
47          immediate await(RS1_EO);
48
49          OUT2 = ((RS1_Q1&!(IN4))|IN3);
50
51          emit(TON0_EI);
52          immediate await(TON0_EO);
53
54          OUT4 = (TIME18==TON0_Q&((RS0_Q1&!(IN4))|IN3)&((RS1_Q1&!(IN4))|
                IN3));
55
56          emit(RS2_EI);
57          immediate await(RS2_EO);
58
59          OUT3 = ((RS2_Q1&!(IN4))|IN3);
60
61          emit(EO); pause;
62      }
63      || RS0:RS(RS0_EI, RS0_EO,
64          IN1,
65          TIME4,
66          RS0_Q1);
67      || RS1:RS(RS1_EI, RS1_EO,
68          IN1,
69          TIME9,
70          RS1_Q1);
71      || TON0:TON(TON0_EI, TON0_EO, CLK,
72          TIME18,
73          PT1m,
74          TON0_Q, TON0_ET);
75      || RS2:RS(RS2_EI, RS2_EO,
76          IN2,
77          TIME14,
78          RS2_Q1);
79  }
```

Listing F.26: *Quartz Model: FBD_SPORTS_HALL*

```
1   module FBD_THER_CODE(
2       event bool ?EI,
3       event bool !EO,
4       bool ?IN1,
5       bool ?IN2,
6       bool ?IN3,
7       bool ?IN4,
8       bool !OUT1,
9       bool !OUT2,
10      bool !OUT3){
11
12      loop{
13          immediate await(EI);
14
15          OUT1 = (IN1|IN2|IN3);
16          OUT2 = ((IN1&IN2&!(IN3))|(IN1&!(IN2)&IN3)|(!(IN1)&IN2&IN3)|(IN1&
                IN2&IN3));
17          OUT3 = (IN1&IN2&IN3);
18
19          emit(EO); pause;
20      }
21  }
```

Listing F.27: *Quartz Model: FBD_THER_CODE*

```
1   module FBD_TOGGLE_SWITCH(
2       event bool ?EI,
3       event bool !EO,
```

```
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool !OUT1){

    loop{
        immediate await(EI);

        OUT1 = (((IN1&!(IN2)&!(IN3)&!(IN4))|(!(IN1)&IN2&!(IN3)&!(IN4))
            |(!(IN1)&!(IN2)&IN3&!(IN4))|(IN1&IN2&IN3&!(IN4)))|((!(IN1)
            &!(IN2)&!(IN3)&IN4)|(IN1&IN2&!(IN3)&IN4)|(IN1&!(IN2)&IN3&IN4
            )|(!(IN1)&IN2&IN3&IN4)));

        emit(EO); pause;
    }
}
```

Listing F.28: *Quartz Model: FBD_TOGGLE_SWITCH*

```
module FBD_VENT_CTRL(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool !OUT1,
    bool !OUT2,
    bool !OUT3){

    loop{
        immediate await(EI);

        OUT1 = (!(IN1)&IN2&IN3&IN4);
        OUT2 = ((IN2&!(IN4)&!(IN1))|(IN4&!(IN3)&!(IN1))|(!(IN3)&IN2&!(
            IN4))|(!(IN1)&IN3&!(IN2))|(!(IN4)&IN1&!(IN2))|(!(IN3)&IN1&!(
            IN2)));
        OUT3 = ((IN1&IN3&IN4)|(IN1&IN2&IN3)|(IN1&IN2&IN4));

        emit(EO); pause;
    }
}
```

Listing F.29: *Quartz Model: FBD_VENT_CTRL*

```
module FBD_WIND_DIR(
    event bool ?EI,
    event bool !EO,
    bool ?IN1,
    bool ?IN2,
    bool ?IN3,
    bool ?IN4,
    bool !OUT1,
    bool !OUT2,
    bool !OUT3,
    bool !OUT4){

    loop{
        immediate await(EI);

        OUT1 = ((IN1&!(IN2)&!(IN3))|(!(IN1)&!(IN2)&!(IN3))|(!(IN1)&!(IN2
            )&IN3));
        OUT2 = ((!(IN1)&!(IN2)&IN3)|(IN1&!(IN2)&IN3)|(IN1&IN2&IN3));
        OUT3 = ((IN1&IN2&IN3)|(!(IN1)&IN2&IN3)|(!(IN1)&IN2&!(IN3)));
```

```
19          OUT4 = ((!(IN1)&IN2&!(IN3))|(IN1&IN2&!(IN3))|(IN1&!(IN2)&!(IN3))
                );
20
21          emit(EO); pause;
22      }
23  }
```

Listing F.30: *Quartz Model:* `FBD_WIND_DIR`

Appendix G

Resulting Data-Flow Oriented SCCharts

```
import "MOVE_bool.sctx"

scchart FBD_TWO_OF_THREE{
  input bool EI
  output bool EO
  input bool xB1_Temp
  input bool xB2_Temp
  input bool xB3_Temp
  output bool xP1_Temp
  ref MOVE_bool MOVE_01

  dataflow:
    MOVE_01 = {EI, ((xB1_Temp&xB2_Temp)|(xB1_Temp&xB3_Temp)|(xB2_Temp&
        xB3_Temp))};
    xP1_Temp = MOVE_01.OUT;
    EO = MOVE_01.EO;
}
```

Listing G.1: *SCChart: FBD_TWO_OF_THREE*

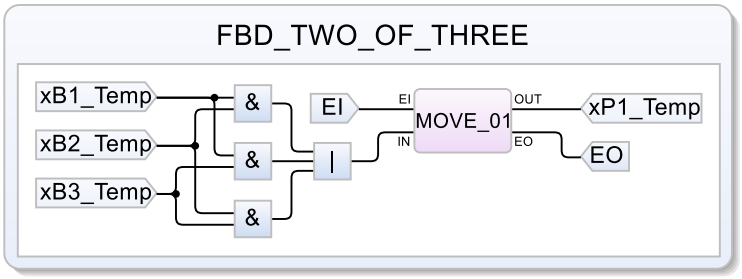


Figure G.1.: *Visualized data-flow oriented SCChart: FBD_TWO_OF_THREE*

```
import "RS.sctx"
import "MOVE_bool.sctx"

scchart FBD_AIR_COND_CTRL{
  input bool EI
  output bool EO
```

```
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  input bool IN5
  input bool IN6
  input bool IN7
  input bool IN8
  input bool IN9
  output bool OUT1
  output bool OUT2
  ref RS RS0
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02

  dataflow:
    RS0={EI, (IN1|IN2|IN3|IN4),((!(IN5)|!(IN6)|!(IN7)|!(IN8))|!(IN9))};
    MOVE_01 = {RS0.EO, RS0.Q1};
    OUT1=MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, !(IN9)};
    OUT2=MOVE_02.OUT;
    EO = MOVE_02.EO;
}
```

Listing G.2: *SCChart: FBD_AIR_COND_CTRL*

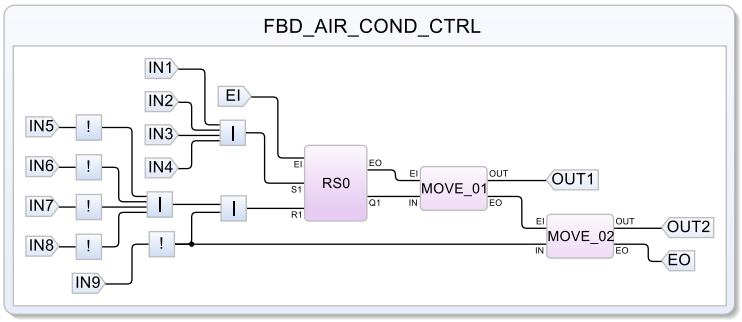


Figure G.2.: *Visualized data-flow oriented SCChart: FBD_AIR_COND_CTRL*

```
import "MOVE_bool.sctx"

scchart FBD_ALARM{
  input bool EI
  output bool EO
  input bool xSENSOR_L
  input bool xSENSOR_M
  input bool xSENSOR_R
  output bool FBD_ALARM
  ref MOVE_bool MOVE_01

  dataflow:
    MOVE_01 = {EI, (!(xSENSOR_L)&!(xSENSOR_M)&!(xSENSOR_R))|(xSENSOR_L&
        xSENSOR_R)};
    FBD_ALARM = MOVE_01.OUT;
    EO = MOVE_01.EO;
}
```

Listing G.3: *SCChart: FBD_ALARM*

```
import "MOVE_bool.sctx"

```

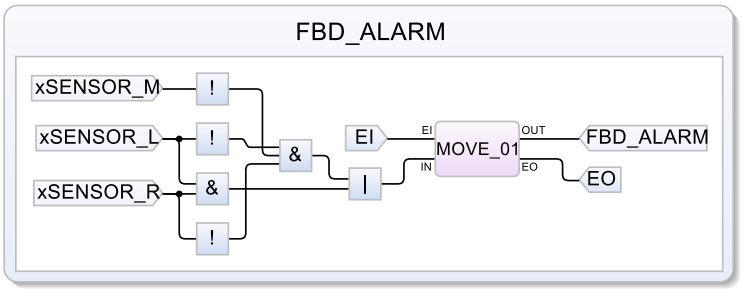


Figure G.3.: *Visualized data-flow oriented SCChart:* `FBD_ALARM`

```
scchart FBD_ANTIVALENCE{
  input bool EI
  output bool EO
  input bool IN0
  input bool IN1
  input bool IN2
  output bool OUT1
  ref MOVE_bool MOVE_01

  dataflow:
    MOVE_01 = {EI, ((!(IN0)&!(IN1)&IN2)|(!(IN0)&!(IN2)&IN1)|(!(IN1)&!(
        IN2)&IN0))};
    OUT1 = MOVE_01.EO;
    EO = MOVE_01.EO;
 }
```

Listing G.4: *SCChart:* `FBD_ANTIVALENCE`

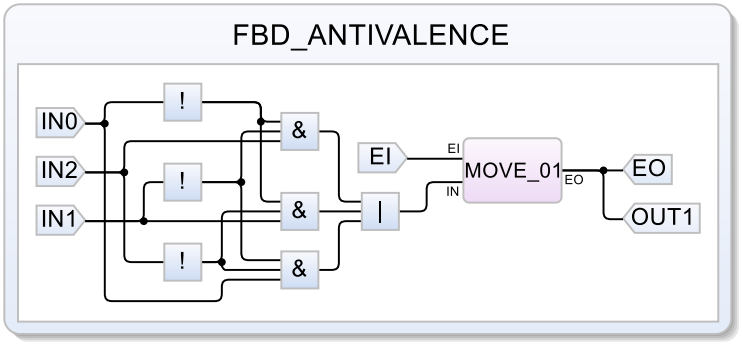


Figure G.4.: *Visualized data-flow oriented SCChart:* `FBD_ANTIVALENCE`

```
import "MOVE_float.sctx"
import "MOVE_int.sctx"

scchart FBD_OP_ARITH{
  input bool EI
  output bool EO
  float x01
  float x02
  float x03
  float x04
  float x05
  float x06
  float x1 = 1.0
  float x2 = 2.0
```

```
  int x3 = 1
  int x4 = 2
  ref MOVE_float MOVE_01
  ref MOVE_float MOVE_02
  ref MOVE_float MOVE_03
  ref MOVE_float MOVE_04
  ref MOVE_int MOVE_05

  dataflow:
    MOVE_01 = {EI, x1 + x2};
    x01 = MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, x1 - x2};
    x02 = MOVE_02.OUT;
    MOVE_03 = {MOVE_02.EO, x1 * x2};
    x03 = MOVE_03.OUT;
    MOVE_04 = {MOVE_03.EO, x1 / x2};
    x04 = MOVE_04.OUT;
    MOVE_05 = {MOVE_04.EO, x3 % x4};
    x06 = MOVE_05.OUT;
    EO = MOVE_05.EO;
}
```

Listing G.5: *SCChart:* `FBD_OP_ARITH`

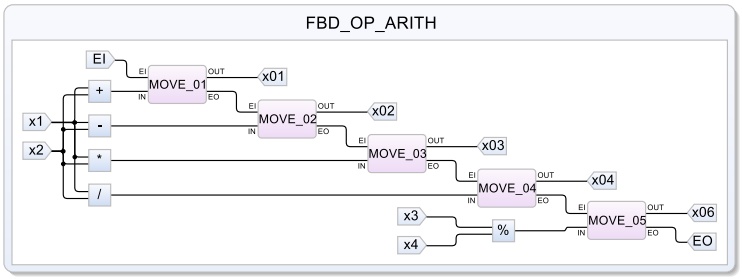


Figure G.5.: *Visualized data-flow oriented SCChart:* `FBD_OP_ARITH`

```
import "MOVE_bool.sctx"
import "TON.sctx"
import "TOF.sctx"
import "RS.sctx"

scchart FBD_BENDING{
  input bool EI
  output bool EO
  input int CLK
  input bool IN1
  input bool IN2
  input bool IN3
  output bool OUT1
  output bool OUT2
  output bool OUT3
  int PT5s
  int PT05s
  int PT1s
  ref TON TON0
  ref TON TON1
  ref TON TON2
  ref TOF TOF0
  ref TON TON3
  ref RS RS0
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02
```

```
27    ref MOVE_bool MOVE_03
28
29    dataflow:
30      TON0 = {EI,CLK,IN1,PT1s};
31      TON1 = {TON0.EO,CLK,(IN1&TON0.Q),PT1s};
32      TON2 = {TON1.EO,CLK, IN2,PT05s};
33      TOF0 = {TON2.EO,CLK, (IN1&TON0.Q),PT1s};
34      MOVE_01 = {TOF0.EO,(IN1&TON0.Q)&!((TON1.Q&TON2.Q))&!((((!(TON2.Q)&
            TOF0.Q)&IN3))};
35      OUT1 = MOVE_01.OUT;
36      MOVE_02 = {MOVE_01.EO, (!((((!(TON2.Q)&TOF0.Q)&IN3))&(!(TON2.Q)&TOF0.
            Q))};
37      OUT2 = MOVE_02.OUT;
38      TON3 = {MOVE_02.EO, CLK,!(IN1),PT5s};
39      RS0 = {TON3.EO,((IN1&TON0.Q)|TON3.Q),(TON1.Q&TON2.Q)};
40      MOVE_03 = {RS0.EO, RS0.Q1};
41      OUT3 = MOVE_03.OUT;
42      EO = MOVE_03.EO;
43  }
```

Listing G.6: *SCChart:* `FBD_BENDING`

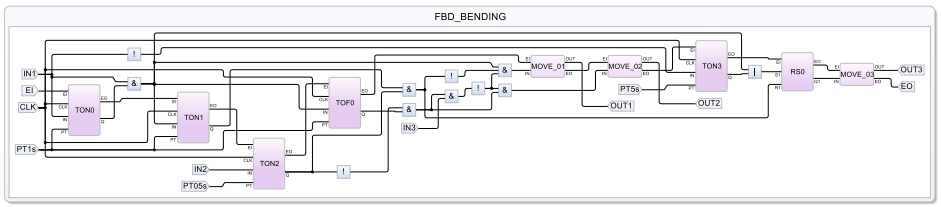


Figure G.6.: *Visualized data-flow oriented SCChart:* `FBD_BENDING`

```
1   import "MOVE_bool.sctx"
2
3   scchart FBD_OP_BOOL{
4     input bool EI
5     output bool EO
6     bool x01
7     bool x02
8     bool x03
9     bool x04
10    bool x1 = true
11    bool x2 = false
12    ref MOVE_bool MOVE_01
13    ref MOVE_bool MOVE_02
14    ref MOVE_bool MOVE_03
15    ref MOVE_bool MOVE_04
16
17    dataflow:
18      MOVE_01 = {EI, !(x1)};
19      x01 = MOVE_01.OUT;
20      MOVE_02 = {MOVE_01.EO, x1 & x2};
21      x02 = MOVE_02.OUT;
22      MOVE_03 = {MOVE_02.EO, x1 | x2};
23      x03 = MOVE_03.OUT;
24      MOVE_04 = {MOVE_03.EO, x1 ^ x2};
25      x04 = MOVE_04.OUT;
26      EO = MOVE_04.EO;
27  }
```

Listing G.7: *SCChart:* `FBD_OP_BOOL`

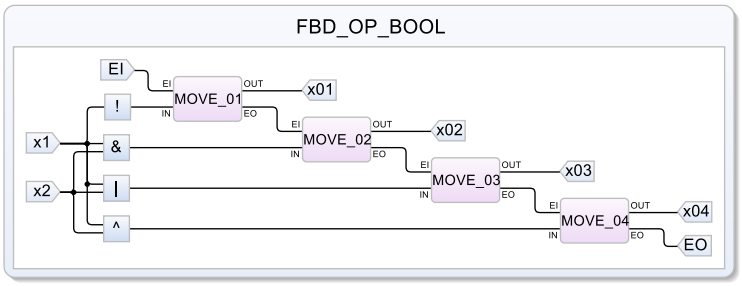


Figure G.7.: *Visualized data-flow oriented SCChart:* `FBD_OP_BOOL`

```
import "MOVE_bool.sctx"

scchart FBD_CYLINDER{
  input bool EI
  output bool EO
  input bool INa1
  input bool INa0
  input bool INS
  input bool INb1
  input bool INc0
  input bool INc1
  input bool INb0
  output bool OUTBp
  output bool OUTAp
  output bool OUTAm
  output bool OUTCp
  bool ME1
  bool ME2
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02
  ref MOVE_bool MOVE_03
  ref MOVE_bool MOVE_04
  ref MOVE_bool MOVE_05
  ref MOVE_bool MOVE_06

  dataflow:
    MOVE_01 = {EI,((ME1|(INa0&INS))&!(INb1))};
    OUTAp=MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO,((ME1|(INa0&INS))&!(INb1))};
    ME1=MOVE_02.OUT;
    MOVE_03 = {MOVE_02.EO, (INa1&((ME1|(INa0&INS))&!(INb1)))};
    OUTBp=MOVE_03.OUT;
    MOVE_04 = {MOVE_03.EO,((ME2|INb1)&!(INc1))};
    ME2=MOVE_04.OUT;
    MOVE_05 = {MOVE_04.EO,(!(((ME1|(INa0&INS))&!(INb1)))&INc0&!(((ME2|
        INb1)&!(INc1))))};
    OUTAm=MOVE_05.OUT;
    MOVE_06 = {MOVE_05.EO,(((ME2|INb1)&!(INc1))&INb0)};
    OUTCp=MOVE_06.OUT;
    EO = MOVE_06.EO;
}
```

Listing G.8: *SCChart:* `FBD_CYLINDER`

```
scchart FBD_DATATYPES{
  input bool EI
  output bool EO
  input bool INS1
  bool A1
  bool A2 = true
```

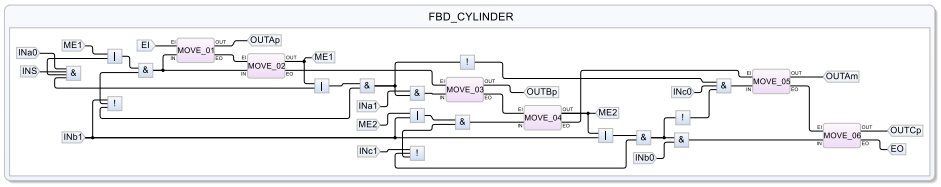


Figure G.8.: *Visualized data-flow oriented SCChart:* `FBD_CYLINDER`

```
  int A5
  int A6 = 2
  int A7
  int A8 = 2
  int A9
  int A10 = 2
  float A13
  float A14 = 1.23
  int A15
  int A16 = 5000
  bool A17[3]
  int A20[3]
  int A21[3]
  int A22[3]
  float A24[3]
  int A25[3]

  dataflow:
}
```

Listing G.9: *SCChart:* `FBD_DATATYPES`

```
import "MOVE_int.sctx"
import "TON.sctx"
import "SR.sctx"

scchart FBD_DEBOUNCE{
  input bool EI
  output bool EO
  input int CLK
  input bool IN
  input int DB_TIME = 2
  output bool OUT
  output int ET_OFF
  ref TON DB_ON
  ref TON DB_OFF
  ref SR DB_FF
  ref MOVE_int MOVE_01
  ref MOVE_int MOVE_02

  dataflow:
    DB_ON = {EI, CLK, IN, DB_TIME};
    DB_OFF = {DB_ON.EO, CLK, !(IN), DB_TIME};
    MOVE_01 = {DB_OFF.EO, DB_OFF.ET};
    ET_OFF = MOVE_01.OUT;
    DB_FF = {MOVE_01.EO, DB_ON.Q, DB_OFF.Q};
    MOVE_02 = {DB_FF.EO, DB_FF.Q1};
    OUT = MOVE_02.OUT;
    EO = MOVE_02.EO;
}
```

Listing G.10: *SCChart:* `FBD_DEBOUNCE`

```
import "MOVE_bool.sctx"
```

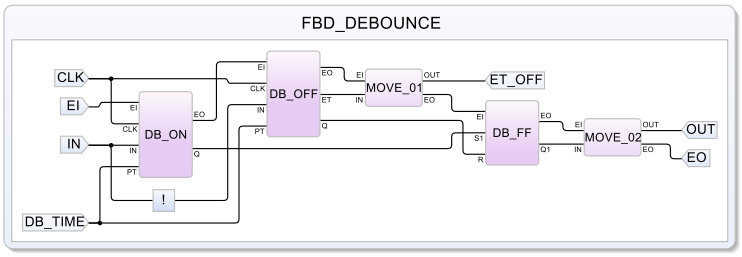


Figure G.9.: *Visualized data-flow oriented SCChart:* `FBD_DEBOUNCE`

```
2
3  scchart FBD_DICE{
4    input bool EI
5    output bool EO
6    input bool IN1
7    input bool IN2
8    input bool IN3
9    input bool IN4
10   output bool OUTa
11   output bool OUTb
12   output bool OUTc
13   output bool OUTd
14   output bool OUTe
15   output bool OUTf
16   output bool OUTg
17   output bool OUTh
18   output bool OUTi
19   ref MOVE_bool MOVE_01
20   ref MOVE_bool MOVE_02
21   ref MOVE_bool MOVE_03
22   ref MOVE_bool MOVE_04
23   ref MOVE_bool MOVE_05
24   ref MOVE_bool MOVE_06
25   ref MOVE_bool MOVE_07
26   ref MOVE_bool MOVE_08
27   ref MOVE_bool MOVE_09
28
29   dataflow:
30     MOVE_01 = {EI,((IN3&!(IN4)&!(IN2))|(IN2&!(IN4)&!(IN1))|(IN4&!(IN2)
         &!(IN3)))};
31     OUTa = MOVE_01.OUT;
32     MOVE_02 = {MOVE_01.EO,((IN2&IN3&!(IN4))|(IN4&!(IN2)&!(IN3)))};
33     OUTb = MOVE_02.OUT;
34     MOVE_03 = {MOVE_02.EO,((IN3&!(IN4))|(IN4&!(IN2)&!(IN3))|(IN1&IN2&!(
         IN4)))};
35     OUTc = MOVE_03.OUT;
36     MOVE_04 = {MOVE_03.EO,((IN4&!(IN2)&!(IN3))|(IN1&IN2&IN3&!(IN4)))};
37     OUTd = MOVE_04.OUT;
38     MOVE_05 = {MOVE_04.EO,((IN1&!(IN4))|(IN1&!(IN3)&!(IN2)))};
39     OUTe = MOVE_05.OUT;
40     MOVE_06 = {MOVE_05.EO,((IN4&!(IN2)&!(IN3))|(IN1&IN2&IN3&!(IN4)))};
41     OUTf = MOVE_06.OUT;
42     MOVE_07 = {MOVE_06.EO,((IN2&IN3&!(IN4))|(IN4&!(IN2)&!(IN3)))};
43     OUTh = MOVE_07.OUT;
44     MOVE_08 = {MOVE_07.EO,((IN3&!(IN4))|(IN4&!(IN2)&!(IN3))|(IN1&IN2&!(
         IN4)))};
45     OUTg = MOVE_08.OUT;
46     MOVE_09 = {MOVE_08.EO,((IN3&!(IN4)&!(IN2))|(IN2&!(IN4)&!(IN1)&!(IN3)
         )|(IN4&!(IN2)&!(IN3)))};
47     OUTi = MOVE_09.OUT;
48     EO = MOVE_09.EO;
```

```
49 }
```

Listing G.11: *SCChart:* `FBD_DICE`

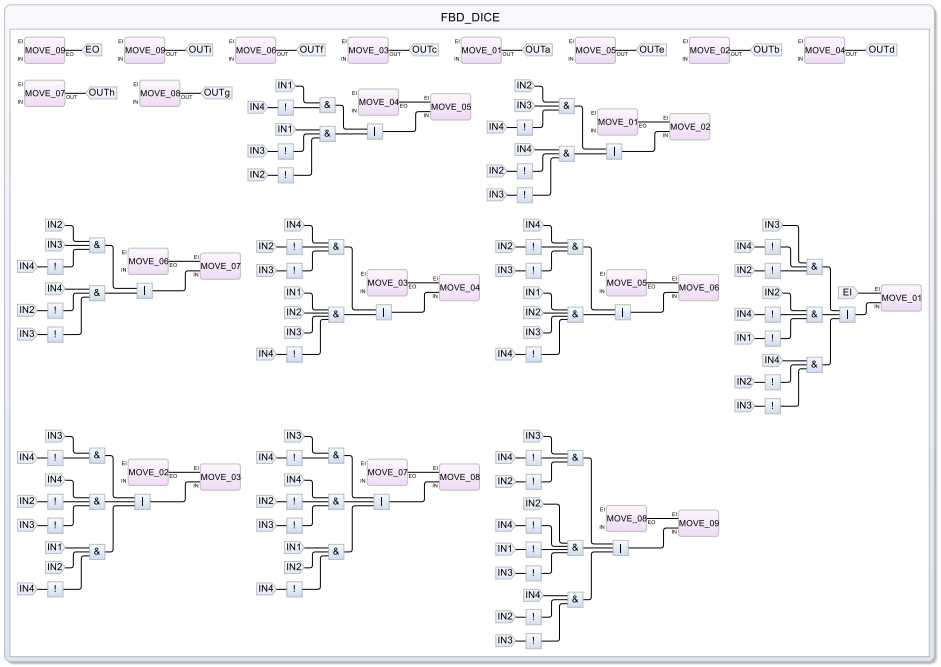


Figure G.10.: *Visualized data-flow oriented SCChart:* `FBD_DICE`

```
1  import "MOVE_bool.sctx"
2
3  scchart FBD_KV_DIAG{
4    input bool EI
5    output bool EO
6    input bool INa
7    input bool INb
8    input bool INc
9    input bool INd
10   output bool OUT1
11   output bool OUT2
12   ref MOVE_bool MOVE_01
13   ref MOVE_bool MOVE_02
14
15   dataflow:
16     MOVE_01 = {EI, ((!(INa)&INb)|(INa&!(INb)&INc))};
17     OUT1 = MOVE_01.OUT;
18     MOVE_02 = {MOVE_01.EO, (((INa==INb&INc)|(!(INa)&!(INc)&INb))&!(INd))
           };
19     OUT2 = MOVE_02.OUT;
20     EO = MOVE_02.EO;
21 }
```

Listing G.12: *SCChart:* `FBD_KV_DIAG`

```
1  import "MOVE_bool.sctx"
2
3  scchart FBD_LEFT_DET{
4    input bool EI
5    output bool EO
```

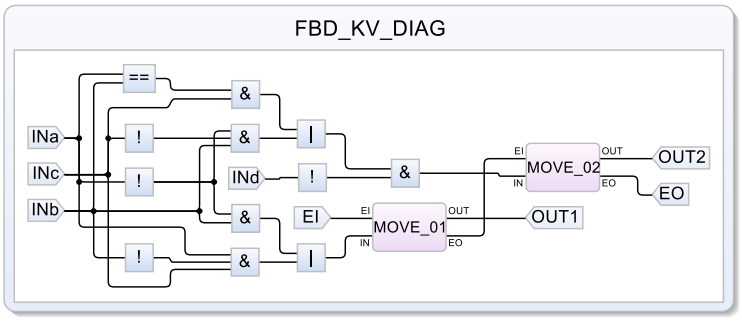


Figure G.11.: *Visualized data-flow oriented SCChart: FBD_KV_DIAG*

```
6     input bool xSENSOR_L
7     input bool xSENSOR_R
8     output bool FBD_LEFT_DET
9     ref MOVE_bool MOVE_01
10
11    dataflow:
12      MOVE_01 = {EI, (xSENSOR_L & !(xSENSOR_R))};
13      FBD_LEFT_DET = MOVE_01.OUT;
14      EO = MOVE_01.EO;
15  }
```

Listing G.13: *SCChart: FBD_LEFT_DET*

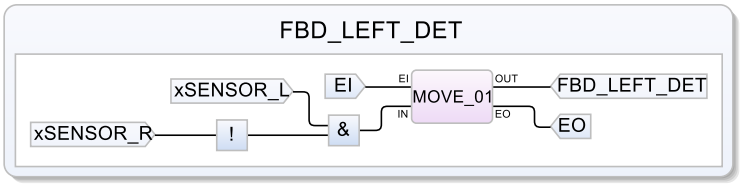


Figure G.12.: *Visualized data-flow oriented SCChart: FBD_LEFT_DET*

```
1   import "MOVE_bool.sctx"
2
3   scchart FBD_POLL{
4     input bool EI
5     output bool EO
6     input bool IN1
7     input bool IN2
8     input bool IN3
9     output bool OUT1
10    output bool OUT2
11    output bool OUT3
12    ref MOVE_bool MOVE_01
13    ref MOVE_bool MOVE_02
14    ref MOVE_bool MOVE_03
15
16    dataflow:
17      MOVE_01 = {EI,((!(IN3)&!(IN2)&IN1)|(!(IN3)&IN2&!(IN1))|(IN3&!(IN2)
            &!(IN1)))};
18      OUT1 = MOVE_01.OUT;
19      MOVE_02 = {MOVE_01.EO, ((!(IN3)&IN2&IN1)|(IN3&!(IN2)&IN1)|(IN3&IN2
            &!(IN1)))};
20      OUT2 = MOVE_02.OUT;
21      MOVE_03 = {MOVE_02.EO, (IN1&IN2&IN3)};
22      OUT3 = MOVE_03.OUT;
```

```
    EO = MOVE_03.EO;
}
```

Listing G.14: *SCChart: FBD_POLL*

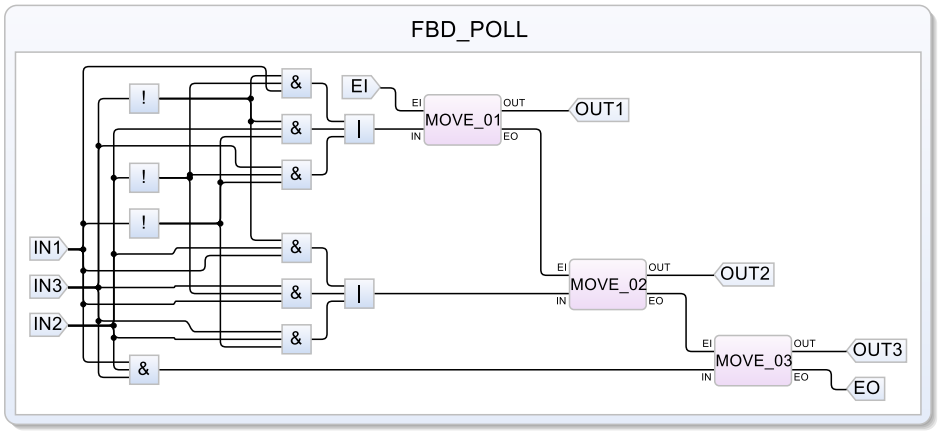


Figure G.13.: *Visualized data-flow oriented SCChart: FBD_POLL*

```
import "MOVE_bool.sctx"

scchart FBD_RES_CTRL1{
  input bool EI
  output bool EO
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  output bool OUTP1
  output bool OUTP2
  output bool OUTP3
  output bool OUTH
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02
  ref MOVE_bool MOVE_03
  ref MOVE_bool MOVE_04

  dataflow:
    MOVE_01 = {EI, ((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1))};
    OUTP1 = MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, ((((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1))|(!(IN3
        )&IN2&!(IN1)))};
    OUTP2 = MOVE_02.OUT;
    MOVE_03 = {MOVE_02.EO, ((!(IN4)&!(IN3)&!(IN2))|(IN4&!(IN3)&IN2)|(!(
        IN3)&IN2&!(IN1))|(!(IN4)&IN3&IN2&IN2))};
    OUTP3 = MOVE_03.OUT;
    MOVE_04 = {MOVE_03.EO, ((IN3&!(IN1))|(IN4&!(IN2)))};
    OUTH = MOVE_04.OUT;
    EO = MOVE_04.EO;
}
```

Listing G.15: *SCChart: FBD_RES_CTRL1*

```
import "MOVE_bool.sctx"

scchart FBD_RES_CTRL2{
  input bool EI
  output bool EO
```

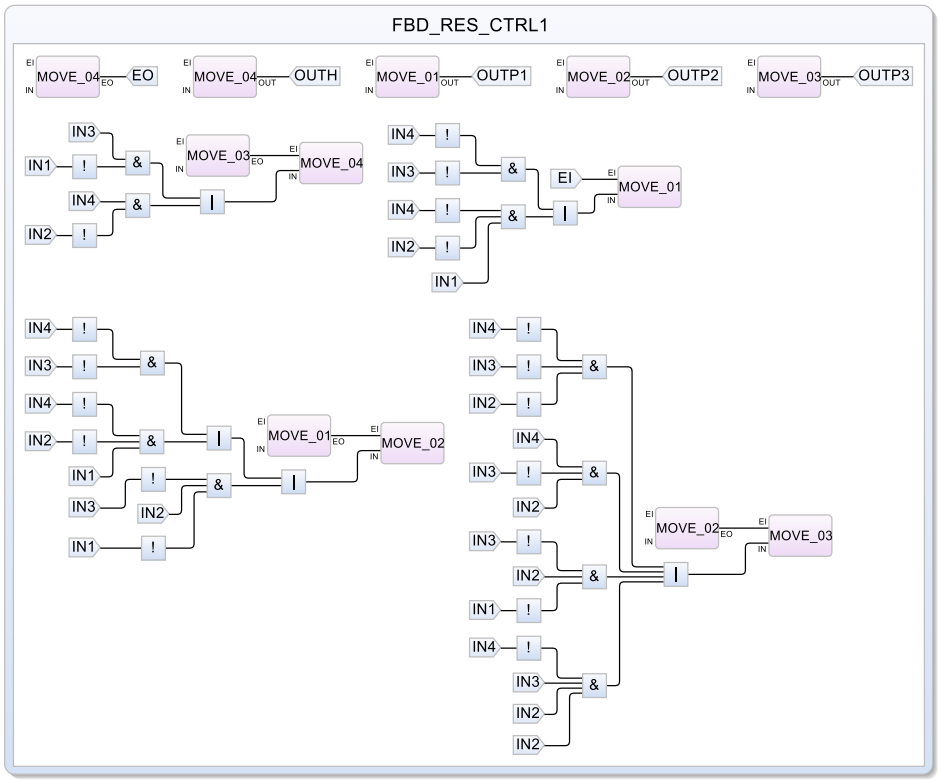


Figure G.14.: *Visualized data-flow oriented SCChart: FBD_RES_CTRL1*

```
6     input bool IN1
7     input bool IN2
8     input bool IN3
9     input bool IN4
10    output bool OUTP1
11    output bool OUTP2
12    output bool OUTP3
13    output bool OUTQ
14    ref MOVE_bool MOVE_01
15    ref MOVE_bool MOVE_02
16    ref MOVE_bool MOVE_03
17    ref MOVE_bool MOVE_04
18
19    dataflow:
20      MOVE_01 = {EI, ((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1))};
21      OUTP3 = MOVE_01.OUT;
22      MOVE_02 = {MOVE_01.EO, (((!(IN4)&!(IN3))|(!(IN4)&!(IN2)&IN1))|(!(IN3
            )&IN2&!(IN1)))};
23      OUTP2 = MOVE_02.OUT;
24      MOVE_03 = {MOVE_02.EO, ((!(IN4)&!(IN3)&!(IN2))|(IN4&!(IN3)&IN2&IN1)
            |(!(IN3)&IN2&!(IN1))|(!(IN4)&IN3&IN2&IN1))};
25      OUTP1 = MOVE_03.OUT;
26      MOVE_04 = {MOVE_03.EO, ((IN3&!(IN1))|(IN4&!(IN2)))};
27      OUTQ = MOVE_04.OUT;
28      EO = MOVE_04.EO;
29  }
```

Listing G.16: *SCChart: FBD_RES_CTRL2*

```
1   import "MOVE_bool.sctx"
2
```

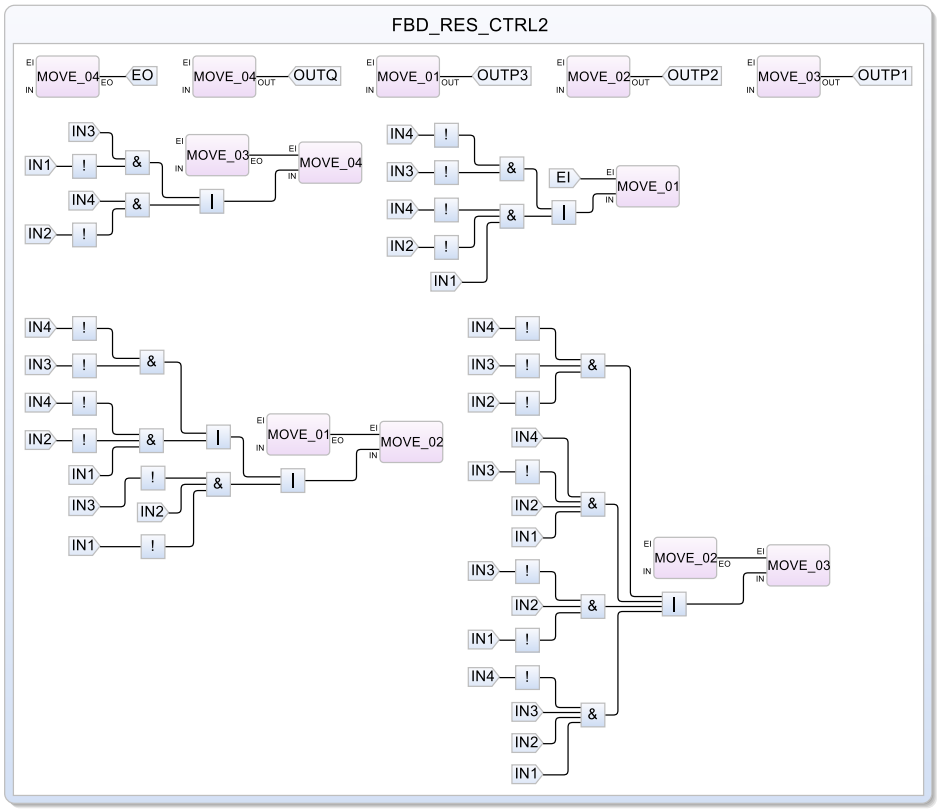


Figure G.15.: *Visualized data-flow oriented SCChart: FBD_RES_CTRL2*

```
scchart FBD_ROLL_DOWN{
  input bool EI
  output bool EO
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  input bool IN5
  input bool IN6
  input bool TIME14
  input bool TIME4
  output bool OUT1
  output bool OUT2
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02

  dataflow:
    MOVE_01 = {EI, (((TIME4&IN6)|(!(IN6)&IN2))&IN4)};
    OUT1 = MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, ((((TIME4&IN6)|(!(IN6)&IN2))&IN4)==(IN5&((IN6
        &IN1&!(TIME14))|(IN3&!(IN6))))&(IN5&((IN6&IN1&!(TIME14))|(IN3&!(
        IN6)))))};
    OUT2 = MOVE_02.OUT;
    EO = MOVE_02.EO;
}
```

Listing G.17: *SCChart: FBD_ROLL_DOWN*

```
import "MOVE_bool.sctx"

```

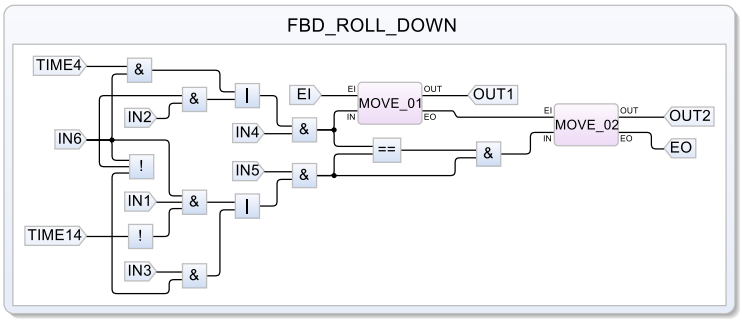


Figure G.16.: *Visualized data-flow oriented SCChart: FBD_ROLL_DOWN*

```
scchart FBD_CABLE_WINCH{
  input bool EI
  output bool EO
  input bool INS1
  input bool INB1
  input bool INB2
  output bool OUT1
  output bool OUT2
  bool ME1
  bool ME2
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02
  ref MOVE_bool MOVE_03
  ref MOVE_bool MOVE_04

  dataflow:
    MOVE_01 = {EI, ((ME1|INS1|!(INB2))&!((!(INB1)|!(INS1)|ME2)))};
    OUT1 = MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, ((ME1|INS1|!(INB2))&!((!(INB1)|!(INS1)|ME2)))
        };
    ME1 = MOVE_02.OUT;
    MOVE_03 = {MOVE_02.EO, ((ME2|!(INB1))&!((!(INB2)|!(INS1)|ME1)))};
    OUT2 = MOVE_03.OUT;
    MOVE_04 = {MOVE_03.EO, ((ME2|!(INB1))&!((!(INB2)|!(INS1)|ME1)))};
    ME2 = MOVE_04.OUT;
    EO = MOVE_04.EO;
}
```

Listing G.18: *SCChart: FBD_CABLE_WINCH*

```
import "MOVE_bool.sctx"

scchart FBD_SEVEN_SEG{
  input bool EI
  output bool EO
  input bool IN0
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  output bool OUTa
  output bool OUTb
  output bool OUTc
  output bool OUTd
  output bool OUTe
  output bool OUTf
  output bool OUTg
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02
```

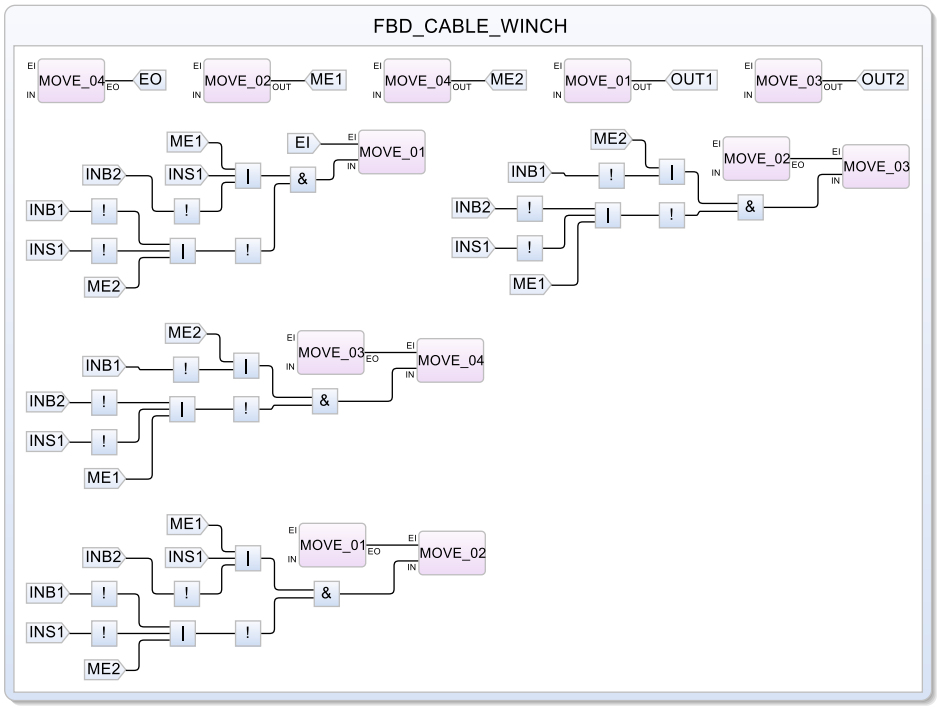


Figure G.17.: *Visualized data-flow oriented SCChart: FBD_CABLE_WINCH*

```
  ref MOVE_bool MOVE_03
  ref MOVE_bool MOVE_04
  ref MOVE_bool MOVE_05
  ref MOVE_bool MOVE_06
  ref MOVE_bool MOVE_07

  dataflow:
    MOVE_01 = {EI, ((!(IN3)&IN1)|(IN2&IN1)|(!(IN2)&!(IN1)&IN3)|(!(IN3)
        &!(IN2)&!(IN0))|(!(IN3)&IN1&IN0))};
    OUTa = MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, ((!(IN3)&!(IN2))|(!(IN2)&!(IN1))|(!(IN3)&IN1&
        IN0)|(IN3&!(IN1)&IN0)|(!(IN3)&!(IN1)&!(IN0)))};
    OUTb = MOVE_02.OUT;
    MOVE_03 = {MOVE_02.EO, ((!(IN3)&!(IN1))|(!(IN3)&IN0)|(!(IN2)&IN3)
        |(!(IN3)&IN2)|(!(IN1)&IN0))};
    OUTc = MOVE_03.OUT;
    MOVE_04 = {MOVE_03.EO, ((!(IN2)&IN1)|(!(IN2)&!(IN0))|(IN1&!(IN0))|(
        IN3&!(IN1)&IN2)|(IN2&!(IN1)&IN0))};
    OUTd = MOVE_04.OUT;
    MOVE_05 = {MOVE_04.EO, ((!(IN2)&!(IN0))|(IN3&IN1)|(IN3&IN2)|(IN2&!(
        IN0)&IN1))};
    OUTe = MOVE_05.OUT;
    MOVE_06 = {MOVE_05.EO, ((IN3&IN2&IN1)|(!(IN3)&IN2&!(IN1))|(!(IN3)&
        IN2&!(IN0))|(IN3&!(IN2)&IN0)|(!(IN2)&!(IN1)&!(IN0)))};
    OUTf = MOVE_06.OUT;
    MOVE_07 = {MOVE_06.EO, (IN3|(!(IN0)&IN2)|(!(IN2)&IN0)|(!(IN1)&IN2))
        };
    OUTg = MOVE_07.OUT;
    EO = MOVE_07.EO;
}
```

Listing G.19: *SCChart: FBD_SEVEN_SEG*

```
import "MOVE_bool.sctx"
```

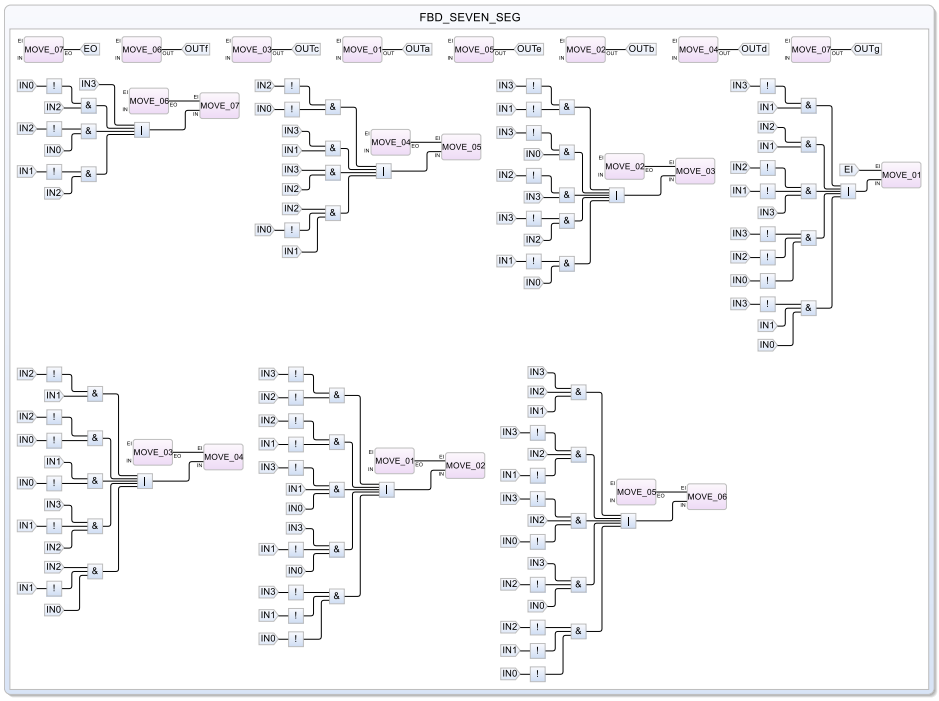


Figure G.18.: *Visualized data-flow oriented SCChart: FBD_SEVEN_SEG*

```
2  import "TOF.sctx"
3
4  scchart FBD_SHOP_WINDOW{
5    input bool EI
6    output bool EO
7    input int CLK
8    input bool IN1
9    input bool IN2
10   input bool IN3
11   input bool IN4
12   input bool TIME3
13   output bool OUT1
14   output bool OUT2
15   output bool OUT3
16   output bool OUT4
17   int PT1m
18   ref TOF TOF0
19   ref TOF TOF1
20   ref MOVE_bool MOVE_01
21   ref MOVE_bool MOVE_02
22   ref MOVE_bool MOVE_03
23   ref MOVE_bool MOVE_04
24
25   dataflow:
26     TOF0 = {EI, CLK, IN3, PT1m};
27     MOVE_01 = {TOF0.EO, ((TIME3|TOF0.Q)&IN2)};
28     OUT1 = MOVE_01.OUT;
29     MOVE_02 = {MOVE_01.EO, (IN2&((((TIME3|TOF0.Q)&IN2)&IN1)|TOF0.Q))};
30     OUT2 = MOVE_02.OUT;
31     MOVE_03 = {MOVE_02.EO, (IN2&((IN1&!(((TIME3|TOF0.Q)&IN2)))|TOF0.Q))
           };
32     OUT3 = MOVE_03.OUT;
33     TOF1 = {MOVE_03.EO, CLK, IN4, PT1m};
34     MOVE_04 = {TOF1.EO, (IN2&(TOF0.Q|TOF1.Q))};
```

```
35      OUT4 = MOVE_04.OUT;
36      EO = MOVE_04.EO;
37  }
```

Listing G.20: *SCChart: FBD_SHOP_WINDOW*

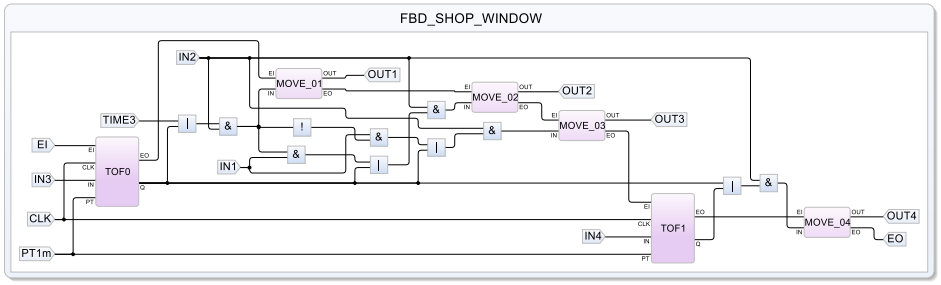


Figure G.19.: *Visualized data-flow oriented SCChart: FBD_SHOP_WINDOW*

```
1  import "MOVE_bool.sctx"
2
3  scchart FBD_SILO_VALVE{
4    input bool EI
5    output bool EO
6    input bool IN1
7    input bool IN2
8    input bool IN3
9    input bool IN4
10   output bool OUT1
11   ref MOVE_bool MOVE_01
12
13   dataflow:
14     MOVE_01 = {EI, ((!(IN3)&!(IN2)&IN1)|(!(IN3)&IN2&!(IN1))|(IN3&!(IN2)
           &!(IN1))|(IN3&IN2&IN1))};
15     OUT1 = MOVE_01.OUT;
16     EO = MOVE_01.EO;
17 }
```

Listing G.21: *SCChart: FBD_SILO_VALVE*

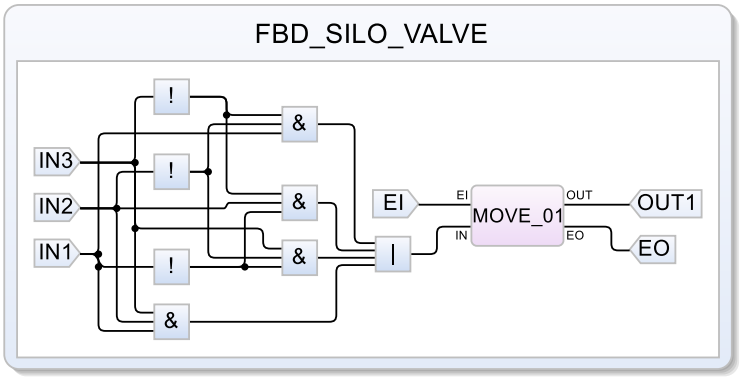


Figure G.20.: *Visualized data-flow oriented SCChart: FBD_SILO_VALVE*

```
1  import "MOVE_int.sctx"
2  import "MOVE_float.sctx"
3
```

```
4  scchart FBD_SIMPLE_FUN{
5    input bool EI
6    output bool EO
7    input float A1
8    input float B1
9    input float C1 = 1.0
10   input output int COUNT
11   output float FBD_SIMPLE_FUN
12   int COUNTP1
13   ref MOVE_float MOVE_01
14   ref MOVE_int MOVE_02
15   ref MOVE_float MOVE_03
16
17   dataflow:
18     MOVE_01 = {EI, COUNT+1};
19     COUNTP1 = MOVE_01.OUT;
20     MOVE_02 = {MOVE_01.EO, COUNTP1};
21     COUNT = MOVE_02.OUT
22     MOVE_03 = {MOVE_02.EO, (A1*B1)/C1};
23     FBD_SIMPLE_FUN = MOVE_03.OUT
24     EO = MOVE_03.EO;
25 }
```

Listing G.22: *SCChart:* `FBD_SIMPLE_FUN`

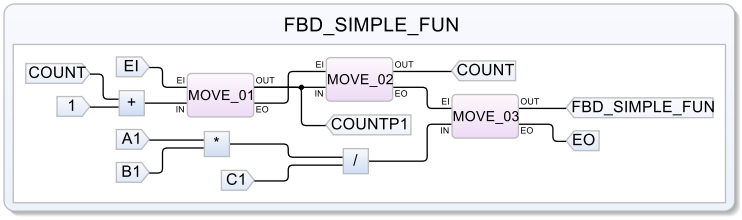


Figure G.21.: *Visualized data-flow oriented SCChart:* `FBD_SIMPLE_FUN`

```
1  import "MOVE_bool.sctx"
2  import "MOVE_int.sctx"
3  import "MOVE_float.sctx"
4  import "FBD_DEBOUNCE.sctx"
5  import "FBD_SIMPLE_FUN.sctx"
6
7  scchart FBD_SIMPLE_PRG1{
8    input bool EI
9    output bool EO
10   input bool PRG_IN
11   output bool PRG_OUT1
12   output float PRG_OUT2
13   output int PRG_ET_OFF
14   int PRG_COUNT = 4
15   float PRG_A
16   float PRG_B
17   float PRG_C
18
19   ref FBD_DEBOUNCE DEBOUNCE_01
20   ref FBD_SIMPLE_FUN FBD_SIMPLE_FUN_11
21   ref MOVE_bool MOVE_01
22   ref MOVE_int MOVE_02
23   ref MOVE_float MOVE_03
24
25   dataflow:
26     DEBOUNCE_01 = {EI, PRG_IN, 2000};
27     MOVE_01 = {DEBOUNCE_01.EO, DEBOUNCE_01.OUT};
28     PRG_OUT1 = MOVE_01.OUT;
```

```
29      MOVE_02 = {MOVE_01.EO, DEBOUNCE_01.ET_OFF};
30      PRG_ET_OFF = MOVE_02.OUT;
31      FBD_SIMPLE_FUN_11 = {MOVE_02.EO, PRG_A + 2, PRG_B, PRG_C, PRG_COUNT
            };
32      MOVE_03 = {FBD_SIMPLE_FUN_11.EO, FBD_SIMPLE_FUN_11.FBD_SIMPLE_FUN};
33      PRG_OUT2 = MOVE_03.OUT;
34      EO = MOVE_03.EO;
35  }
```

Listing G.23: *SCChart: FBD_SIMPLE_PRG1*

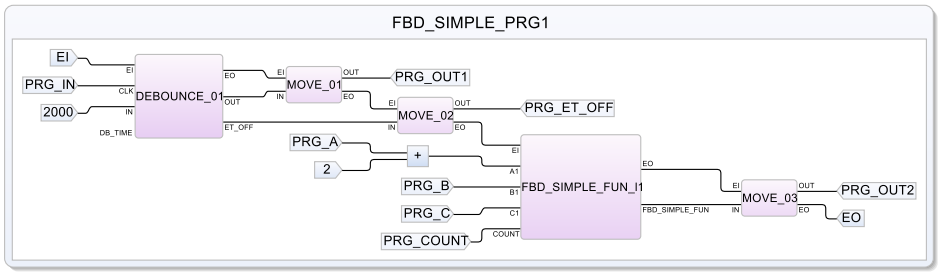


Figure G.22.: *Visualized data-flow oriented SCChart: FBD_SIMPLE_PRG1*

```
1   import "MOVE_int.sctx"
2   import "ST_LOOP_HEAD_SCL.sctx"
3
4   scchart FBD_SIMPLE_PRG2{
5     input bool EI
6     output bool EO
7     int OUT
8     ref ST_LOOP_HEAD_SCL ST_LOOP_HEAD0
9     ref MOVE_int MOVE_01
10
11    dataflow:
12      ST_LOOP_HEAD0 = {EI}
13      MOVE_01 = {ST_LOOP_HEAD0.EO, ST_LOOP_HEAD0.y}
14      OUT = MOVE_01.OUT
15      EO = MOVE_01.EO
16  }
```

Listing G.24: *SCChart: FBD_SIMPLE_PRG2*

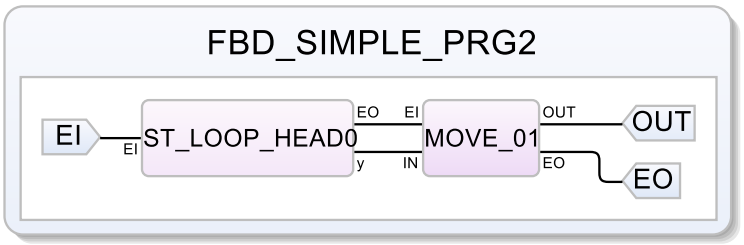


Figure G.23.: *Visualized data-flow oriented SCChart: FBD_SIMPLE_PRG2*

```
1   import "MOVE_bool.sctx"
2
3   scchart FBD_SMOKE_DET{
4     input bool EI
5     output bool EO
6     input bool IN1
```

```
  input bool IN2
  input bool IN3
  input bool IN4
  output bool OUT1
  output bool OUT2
  output bool OUT3
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02
  ref MOVE_bool MOVE_03

  dataflow:
    MOVE_01 = {EI, (!(IN1)|!(IN2)|!(IN3)|!(IN4))};
    OUT1 = MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, ((((!(IN1)&!(IN2))|(!(IN1)&!(IN3)))|(!(IN1)&!(
        IN4)&(!(IN3)&!(IN2))&(!(IN2)&!(IN4))&(!(IN3)&!(IN4))))};
    OUT2 = MOVE_02.OUT;
    MOVE_03 = {MOVE_02.EO, (!(IN1)&!(IN2)&!(IN3)&!(IN4))};
    OUT3 = MOVE_03.OUT;
    EO = MOVE_03.EO;
}
```

Listing G.25: *SCChart:* `FBD_SMOKE_DET`

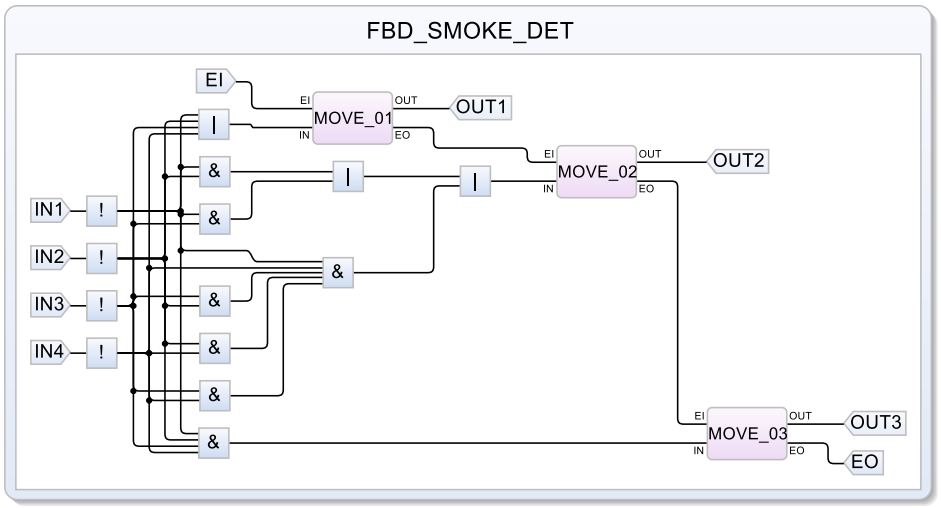


Figure G.24.: *Visualized data-flow oriented SCChart:* `FBD_SMOKE_DET`

```
import "MOVE_bool.sctx"
import "RS.sctx"
import "TON.sctx"

scchart FBD_SPORTS_HALL{
  input bool EI
  output bool EO
  input int CLK
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  input bool TIME4
  input bool TIME9
  input bool TIME14
  input bool TIME18
  output bool OUT1
  output bool OUT2
```

```
19    output bool OUT3
20    output bool OUT4
21    int PT1m
22    ref RS RS0
23    ref RS RS1
24    ref RS RS2
25    ref TON TON0
26    ref MOVE_bool MOVE_01
27    ref MOVE_bool MOVE_02
28    ref MOVE_bool MOVE_03
29    ref MOVE_bool MOVE_04
30
31    dataflow:
32      RS0 = {EI, IN1, TIME4};
33      MOVE_01 = {RS0.EO, ((RS0.Q1&!(IN4))|IN3)};
34      OUT1 = MOVE_01.OUT;
35      RS1 = {MOVE_01.EO, IN1,TIME9};
36      MOVE_02 = {RS1.EO, ((RS1.Q1&!(IN4))|IN3)};
37      OUT2 = MOVE_02.OUT;
38      TON0 = {MOVE_02.EO, CLK, TIME18,PT1m};
39      MOVE_03 = {TON0.EO, (TIME18==TON0.Q&((RS0.Q1&!(IN4))|IN3)&((RS1.Q1
            &!(IN4))|IN3))};
40      OUT4 = MOVE_03.OUT;
41      RS2 = {MOVE_03.EO, IN2,TIME14};
42      MOVE_04 = {RS2.EO, ((RS2.Q1&!(IN4))|IN3)};
43      OUT3 = MOVE_04.OUT;
44      EO = MOVE_04.EO;
45  }
```

Listing G.26: *SCChart:* `FBD_SPORTS_HALL`

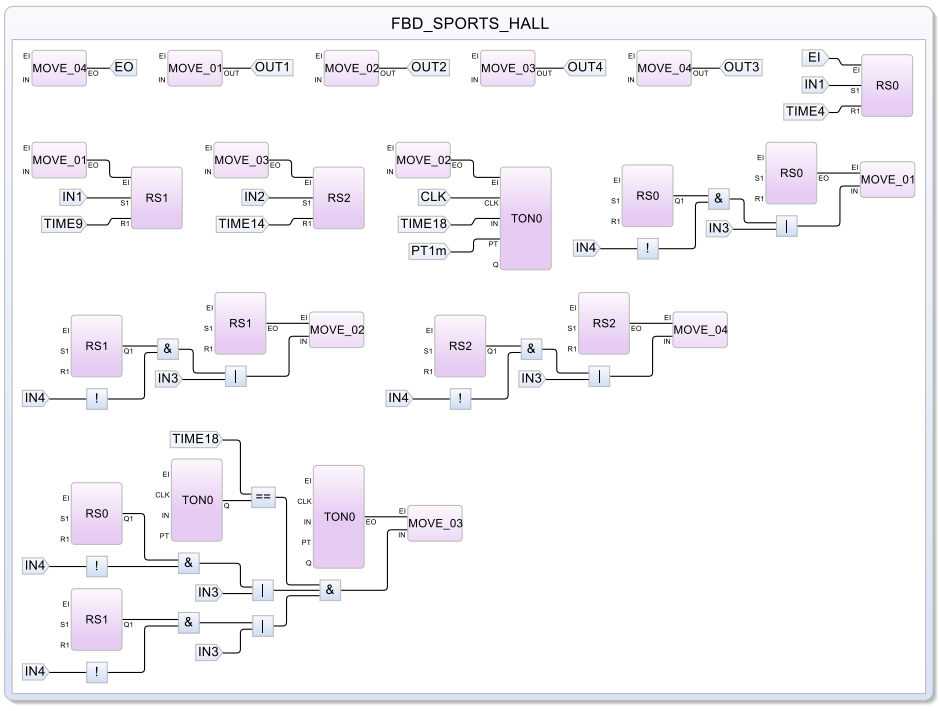


Figure G.25.: *Visualized data-flow oriented SCChart:* `FBD_SPORTS_HALL`

```
1   import "MOVE_bool.sctx"
2
```

```
scchart FBD_THER_CODE{
  input bool EI
  output bool EO
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  output bool OUT1
  output bool OUT2
  output bool OUT3
  ref MOVE_bool MOVE_01
  ref MOVE_bool MOVE_02
  ref MOVE_bool MOVE_03

  dataflow:
    MOVE_01 = {EI, (IN1|IN2|IN3)};
    OUT1 = MOVE_01.OUT;
    MOVE_02 = {MOVE_01.EO, ((IN1&IN2&!(IN3))|(IN1&!(IN2)&IN3)|(!(IN1)&
        IN2&IN3)|(IN1&IN2&IN3))};
    OUT2 = MOVE_02.OUT;
    MOVE_03 = {MOVE_02.EO,(IN1&IN2&IN3)};
    OUT3 = MOVE_03.OUT;
    EO = MOVE_03.EO;
}
```

Listing G.27: *SCChart:* `FBD_THER_CODE`

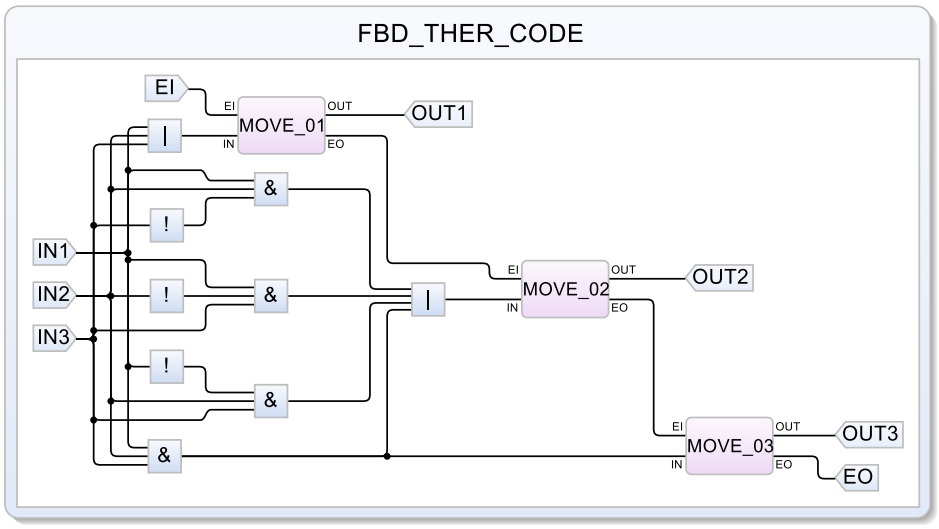


Figure G.26.: *Visualized data-flow oriented SCChart:* `FBD_THER_CODE`

```
import "MOVE_bool.sctx"

scchart FBD_TOGGLE_SWITCH{
  input bool EI
  output bool EO
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  output bool OUT1
  ref MOVE_bool MOVE_01

  dataflow:
```

```
14      MOVE_01 = {EI, (((IN1&!(IN2)&!(IN3)&!(IN4))|(!(IN1)&IN2&!(IN3)&!(IN4
            ))|(!(IN1)&!(IN2)&IN3&!(IN4))|(IN1&IN2&IN3&!(IN4)))|((!(IN1)&!(
            IN2)&!(IN3)&IN4)|(IN1&IN2&!(IN3)&IN4)|(IN1&!(IN2)&IN3&IN4)|(!(
            IN1)&IN2&IN3&IN4)))};
15      OUT1 = MOVE_01.OUT;
16      EO = MOVE_01.EO;
17  }
```

Listing G.28: *SCChart: `FBD_TOGGLE_SWITCH`*

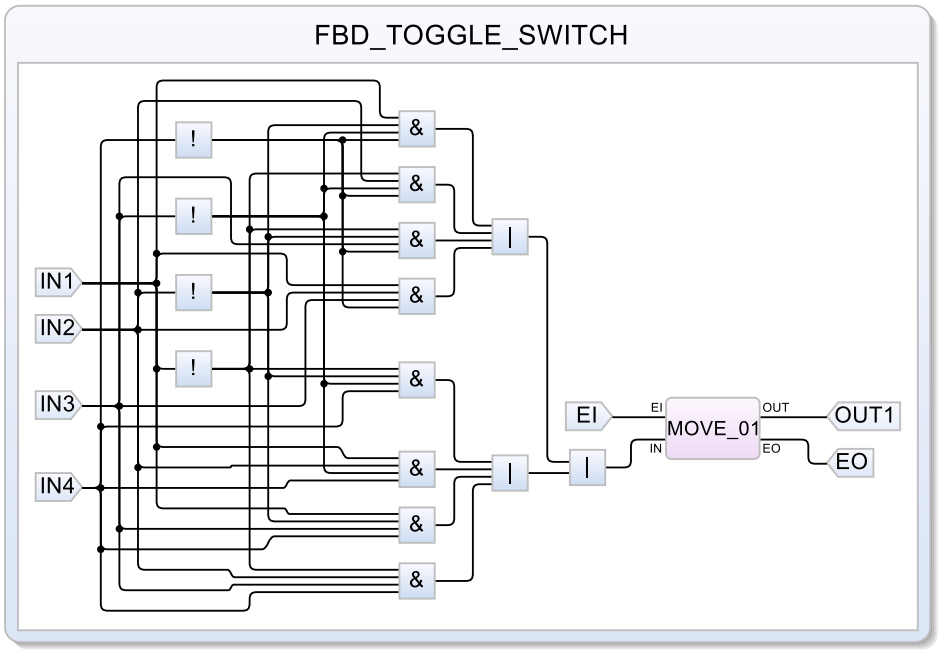


Figure G.27.: *Visualized data-flow oriented SCChart: `FBD_TOGGLE_SWITCH`*

```
1   import "MOVE_bool.sctx"
2
3   scchart FBD_VENT_CTRL{
4     input bool EI
5     output bool EO
6     input bool IN1
7     input bool IN2
8     input bool IN3
9     input bool IN4
10    output bool OUT1
11    output bool OUT2
12    output bool OUT3
13    ref MOVE_bool MOVE_01
14    ref MOVE_bool MOVE_02
15    ref MOVE_bool MOVE_03
16
17    dataflow:
18      MOVE_01 = {EI, (!(IN1)&IN2&IN3&IN4)};
19      OUT1 = MOVE_01.OUT;
20      MOVE_02 = {MOVE_01.EO, ((IN2&!(IN4)&!(IN1))|(IN4&!(IN3)&!(IN1))|(!(
            IN3)&IN2&!(IN4))|(!(IN1)&IN3&!(IN2))|(!(IN4)&IN1&!(IN2))|(!(IN3)
            &IN1&!(IN2)))};
21      OUT2 = MOVE_02.OUT;
22      MOVE_03 = {MOVE_02.EO, ((IN1&IN3&IN4)|(IN1&IN2&IN3)|(IN1&IN2&IN4))};
23      OUT3 = MOVE_03.OUT;
```

```
24      EO = MOVE_03.EO;
25  }
```

Listing G.29: *SCChart:* `FBD_VENT_CTRL`

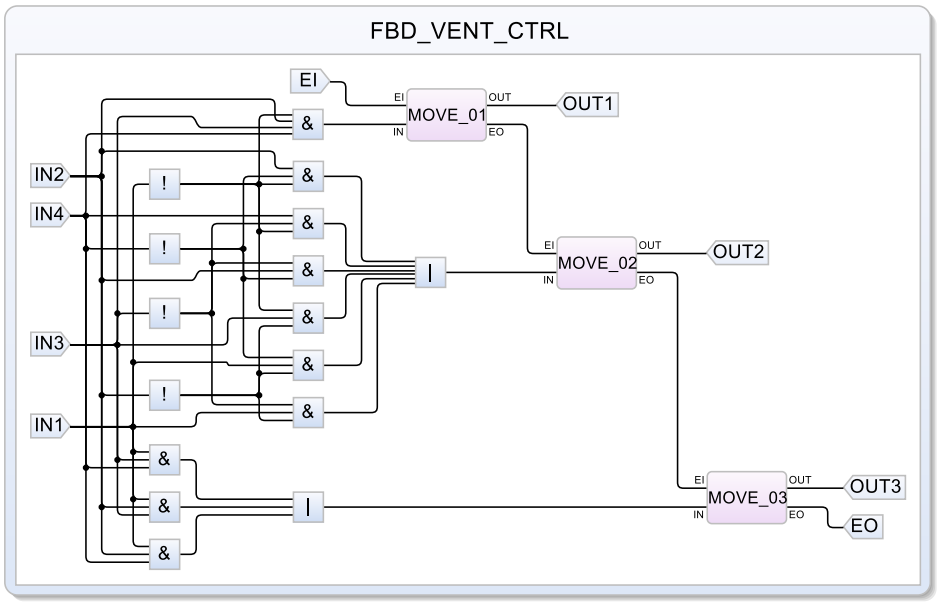


Figure G.28.: *Visualized data-flow oriented SCChart:* `FBD_VENT_CTRL`

```
1   import "MOVE_bool.sctx"
2
3   scchart FBD_WIND_DIR{
4     input bool EI
5     output bool EO
6     input bool IN1
7     input bool IN2
8     input bool IN3
9     output bool OUT1
10    output bool OUT2
11    output bool OUT3
12    output bool OUT4
13    ref MOVE_bool MOVE_01
14    ref MOVE_bool MOVE_02
15    ref MOVE_bool MOVE_03
16    ref MOVE_bool MOVE_04
17
18    dataflow:
19      MOVE_01 = {EI, ((IN1&!(IN2)&!(IN3))|(!(IN1)&!(IN2)&!(IN3))|(!(IN1)
            &!(IN2)&IN3))};
20      OUT1 = MOVE_01.OUT;
21      MOVE_02 = {MOVE_01.EO, ((!(IN1)&!(IN2)&IN3)|(IN1&!(IN2)&IN3)|(IN1&
            IN2&IN3))};
22      OUT2 = MOVE_02.OUT;
23      MOVE_03 = {MOVE_02.EO, ((IN1&IN2&IN3)|(!(IN1)&IN2&IN3)|(!(IN1)&IN2
            &!(IN3)))};
24      OUT3 = MOVE_03.OUT;
25      MOVE_04 = {MOVE_03.EO, ((!(IN1)&IN2&!(IN3))|(IN1&IN2&!(IN3))|(IN1&!(
            IN2)&!(IN3)))};
26      OUT4 = MOVE_04.OUT;
27      EO = MOVE_04.EO;
```

```
28 }
```

Listing G.30: *SCChart:* `FBD_WIND_DIR`

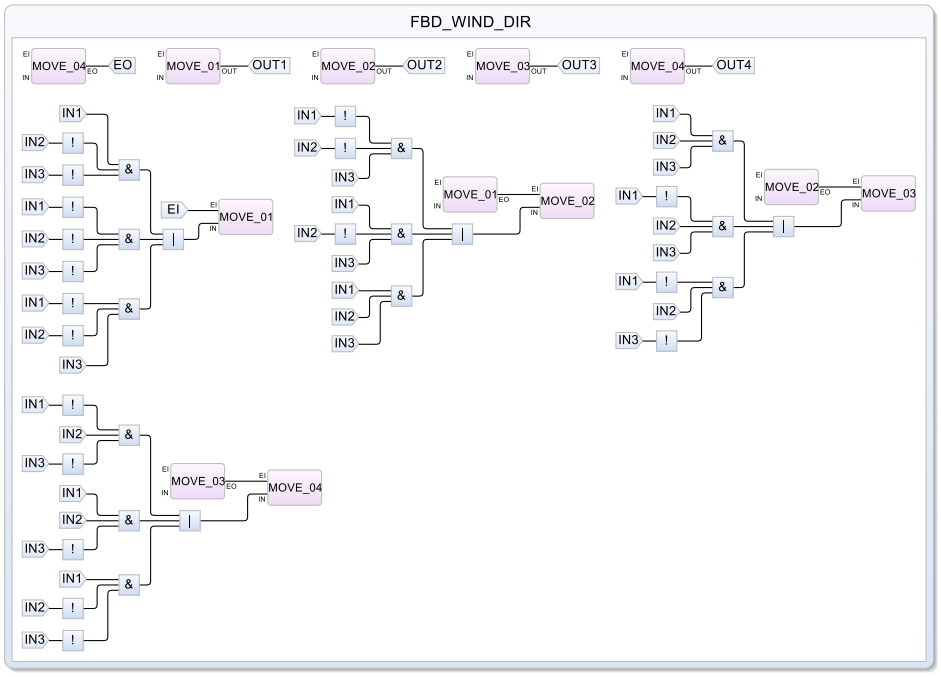


Figure G.29.: *Visualized data-flow oriented SCChart:* `FBD_WIND_DIR`

Appendix H

Resulting Control-Flow Oriented SCCharts

```
scchart ST_ALARM{
 input bool xSENSOR_L
 input bool xSENSOR_M
 input bool xSENSOR_R
 output bool ST_ALARM

 region:
   initial state S0
   immediate do ST_ALARM = (!(xSENSOR_L) & !(xSENSOR_M) & !(xSENSOR_R))
        | (xSENSOR_L & xSENSOR_R) go to S1
   final state S1
}
```

Listing H.1: *SCChart: ST_ALARM*

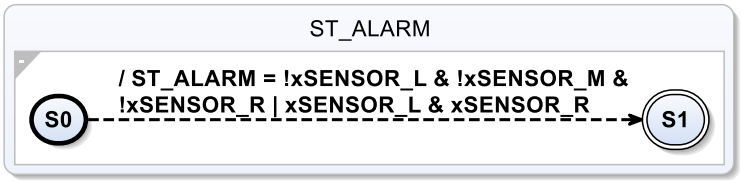


Figure H.1.: *Visualized control-flow oriented SCChart: ST_ALARM*

```
scchart ST_LOOP_FOOT{
 input signal bool EI
 output signal bool EO
 output int y
 int x0
 int x1
 int x2
 int i
 int i0
 int i1

 region:
   initial state S1
   do x0 = 0; x1 = 1; x2 = 2; i0 = 0; i1 = 10 abort to S2

```

```
    state S2{
      initial state S1
      immediate if EI do i = i0 abort to S2

      state S2 {
        initial state S1
        immediate do y = x0 go to S2

        state S2
        do i = pre(i) + pre(x2) abort to S3

        final state S3
        immediate if !(i > i1) go to S1
      }
      immediate if (i > i1) do y = x1; EO join to S3

      final state S3
      abort to S1
    }
}
```

Listing H.2: *SCChart: ST_LOOP_FOOT*

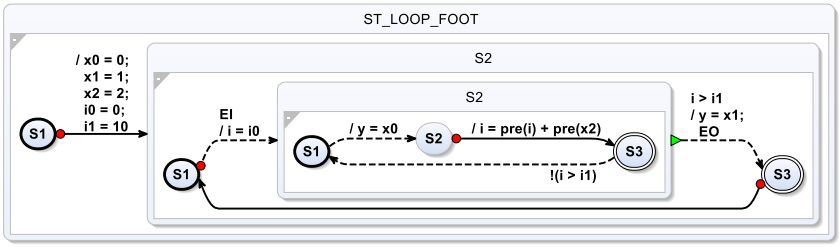


Figure H.2.: *Visualized control-flow oriented SCChart: ST_LOOP_FOOT*

```
scchart ST_LOOP_HEAD{
 input signal bool EI
 output signal bool EO
 output int y
 int x1
 int x2
 int i
 int i0
 int i1

 region:
   initial state S1
   do x1 = 1; x2 = 2; i0 = 0; i1 = 10 abort to S2

   state S2{
     initial state S1
     immediate if EI do i = i0 abort to S2

     state S2 {
       initial final state S1
       immediate if (i <= i1) go to S2

       state S2
       do i = pre(i) + pre(x2) abort to S3

       state S3
       immediate go to S1
     }
```

```
29        immediate if !(i <= i1) do y = x1; EO join to S3
30
31        state S3
32        abort to S1
33      }
34  }
```

Listing H.3: *SCChart: ST_LOOP_HEAD*

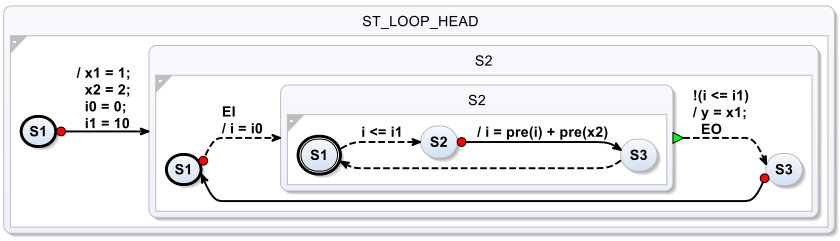


Figure H.3.: *Visualized control-flow oriented SCChart: ST_LOOP_HEAD*

```
1   scchart ST_OP_ARITH{
2    input signal bool EI
3    output signal bool EO
4    float x01
5    float x02
6    float x03
7    float x04
8    float x05
9    float x06
10   float x1
11   float x2
12   int x3
13   int x4
14
15   region:
16     initial state S1
17     immediate do x1 = 1.0; x2 = 2.0; x3 = 1; x4 = 2 abort to S2
18
19     state S2{
20       initial state S1
21       immediate if EI do x01 = x1 + x2; x02 = x1 - x2; x03 = x1 * x2;
             x04 = x1 / x2; x06 = x3 % x4; EO abort to S2
22
23       state S2
24       abort to S1
25     }
26  }
```

Listing H.4: *SCChart: ST_OP_ARITH*

```
1   scchart ST_RS{
2    input signal bool EI
3    output signal bool EO
4    input bool SET
5    input bool RESET1
6    output bool Q1
7    bool Q1_Tmp
8
9    region:
10     initial state S1{
11       initial state S1
12       immediate if EI abort to S2
```

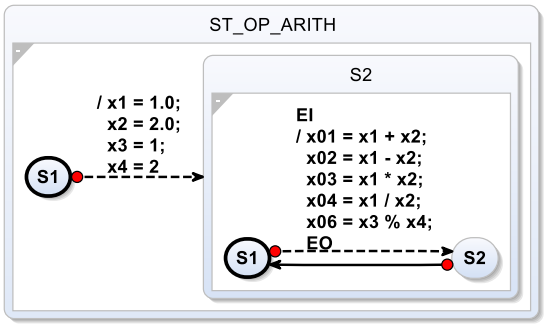


Figure H.4.: *Visualized control-flow oriented SCChart: ST_OP_ARITH*

```
13
14        state S2
15        do Q1_Tmp = (pre(SET) | pre(Q1_Tmp)) & !(pre(RESET1)); Q1 = Q1_Tmp
              ; EO abort to S3
16
17        state S3
18        abort to S1
19      }
20  }
```

Listing H.5: *SCChart: ST_RS*

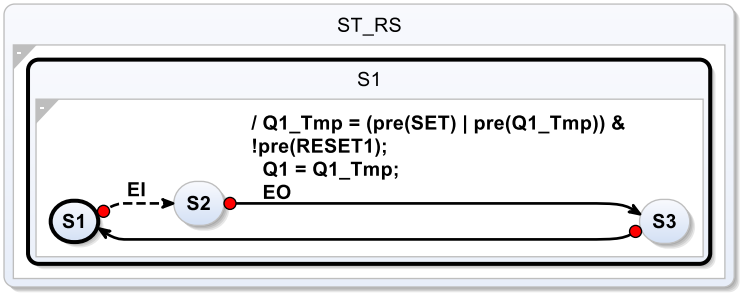


Figure H.5.: *Visualized control-flow oriented SCChart: ST_RS*

```
1   scchart ST_TWO_OF_THREE{
2    input signal bool EI
3    output signal bool EO
4    input bool xB1_Temp
5    input bool xB2_Temp
6    input bool xB3_Temp
7    output bool xP1_Temp
8
9    region:
10     initial state S1{
11       initial state S1
12       immediate if EI do xP1_Temp = ((xB1_Temp&xB2_Temp)|(xB1_Temp&
             xB3_Temp)|(xB2_Temp&xB3_Temp)); EO abort to S2
13
14       state S2
15       abort to S1
16     }
17  }
```

Listing H.6: *SCChart: ST_TWO_OF_THREE*

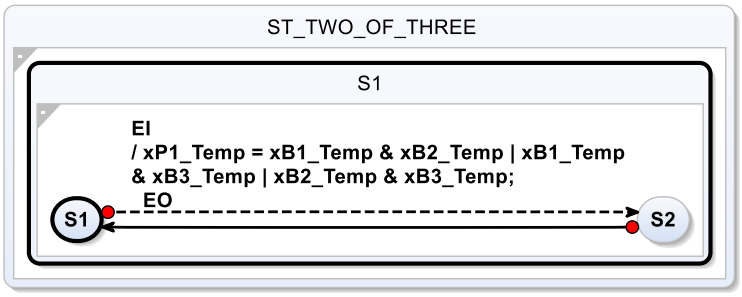


Figure H.6.: *Visualized control-flow oriented SCChart:* `ST_TWO_OF_THREE`

```
scchart FBD_OP_BOOL{
 input signal bool EI
 output signal bool EO
 bool x01
 bool x02
 bool x03
 bool x04
 bool x05
 bool x06
 bool x1
 bool x2

 region:
   initial state S1
   do x1 = true; x2 = false abort to S2

   state S2{
     initial state S1
     immediate if EI do x01 = !(x1); x02 = x1 & x2; x03 = x1 | x2; x04
         = x1 ^ x2; EO abort to S2
     state S2
     abort to S1
   }
}
```

Listing H.7: *SCChart:* `FBD_OP_BOOL`

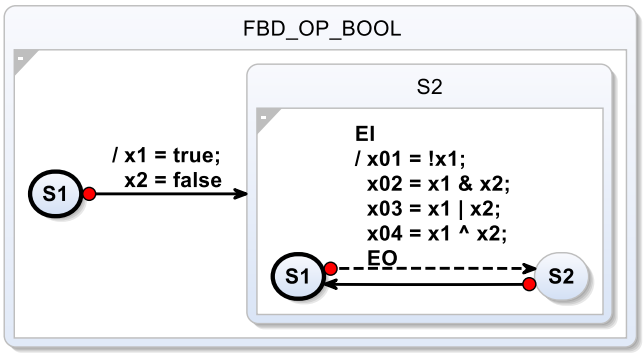


Figure H.7.: *Visualized control-flow oriented SCChart:* `FBD_OP_BOOL`

```
scchart FBD_DATATYPES{
 input signal bool EI
 output signal bool EO
 bool A1
```

```
  bool A2
  int A5
  int A6
  int A7
  int A8
  int A9
  int A10
  float A13
  float A14
  int A15
  int A16
  bool A17[3]
  int A20[3]
  int A21[3]
  int A22[3]
  float A24[3]
  int A25[3]

  region:
    initial state S1
    do A2 = true; A6 = 2; A8 = 2; A10 = 2; A14 = 1.23; A16 = 5000 abort
        to S2

    state S2{
      initial state S1
      immediate if EI do EO abort to S2
      state S2
      abort to S1
    }
}
```

Listing H.8: *SCChart: FBD_DATATYPES*

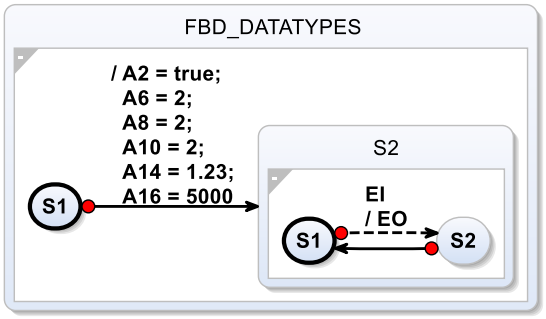


Figure H.8.: *Visualized control-flow oriented SCChart: FBD_DATATYPES*

```
scchart FBD_KV_DIAG{
  input signal bool EI
  output signal bool EO
  input bool INa
  input bool INb
  input bool INc
  input bool INd
  output bool OUT1
  output bool OUT2

  region:
    initial state S1{
      initial state S1
      immediate if EI do OUT1 = ((!(INa)&INb)|(INa&!(INb)&INc)); OUT2 =
          (((INa==INb&INc)|(!(INa)&!(INc)&INb))&!(INd)); EO abort to S2
      state S2
```

```
16        abort to S1
17      }
18  }
```

Listing H.9: *SCChart: FBD_KV_DIAG*

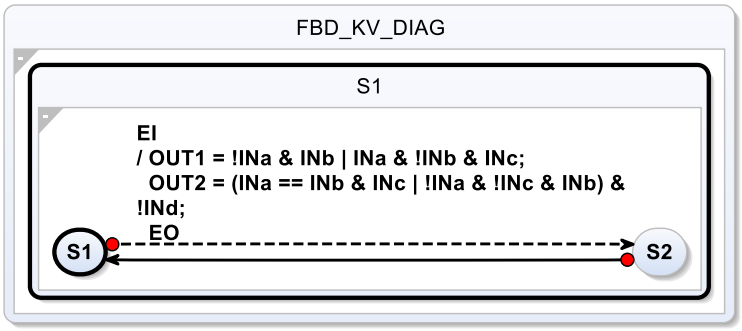


Figure H.9.: *Visualized control-flow oriented SCChart: FBD_KV_DIAG*

```
1   scchart FBD_LEFT_DET{
2    input signal bool EI
3    output signal bool EO
4    input bool xSENSOR_L
5    input bool xSENSOR_R
6    output bool FBD_LEFT_DET
7
8    region:
9      initial state S1{
10       initial state S1
11       immediate if EI do FBD_LEFT_DET = (xSENSOR_L & !(xSENSOR_R)); EO
             abort to S2
12       state S2
13       abort to S1
14     }
15  }
```

Listing H.10: *SCChart: FBD_LEFT_DET*

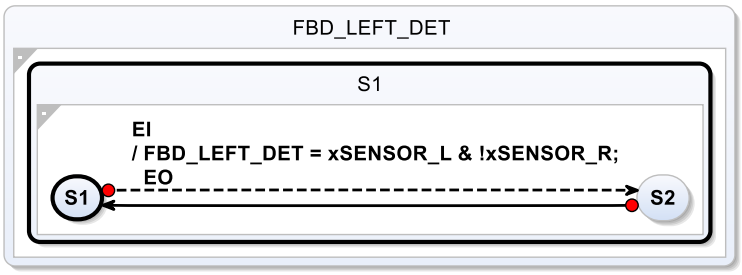


Figure H.10.: *Visualized control-flow oriented SCChart: FBD_LEFT_DET*

```
1   scchart FBD_ROLL_DOWN{
2    input signal bool EI
3    output signal bool EO
4    input bool IN1
5    input bool IN2
6    input bool IN3
7    input bool IN4
```

```
  input bool IN5
  input bool IN6
  input bool TIME14
  input bool TIME4
  output bool OUT1
  output bool OUT2

  region:
    initial state S1{
      initial state S1
      immediate if EI do OUT1 = (((TIME4&IN6)|(!(IN6)&IN2))&IN4); OUT2 =
           ((((TIME4&IN6)|(!(IN6)&IN2))&IN4)==(IN5&((IN6&IN1&!(TIME14))
          |(IN3&!(IN6))))&(IN5&((IN6&IN1&!(TIME14))|(IN3&!(IN6))))); EO
          abort to S2
      state S2
      abort to S1
    }
}
```

Listing H.11: *SCChart: FBD_ROLL_DOWN*

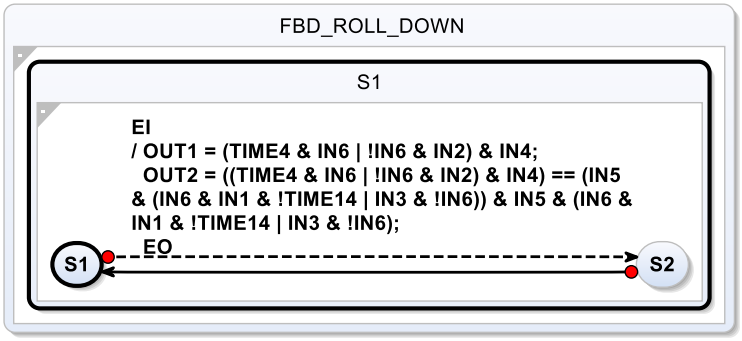


Figure H.11.: *Visualized control-flow oriented SCChart: FBD_ROLL_DOWN*

```
scchart FBD_THER_CODE{
  input signal bool EI
  output signal bool EO
  input bool IN1
  input bool IN2
  input bool IN3
  input bool IN4
  output bool OUT1
  output bool OUT2
  output bool OUT3

  region:
    initial state S1{
      initial state S1
      immediate if EI do OUT1 = (IN1|IN2|IN3); OUT2 = ((IN1&IN2&!(IN3))
          |(IN1&!(IN2)&IN3)|(!(IN1)&IN2&IN3)|(IN1&IN2&IN3)); OUT3 = (IN1
          &IN2&IN3); EO abort to S2
      state S2
      abort to S1
    }
}
```

Listing H.12: *SCChart: FBD_THER_CODE*

```
scchart FBD_TOGGLE_SWITCH{
  input signal bool EI
```

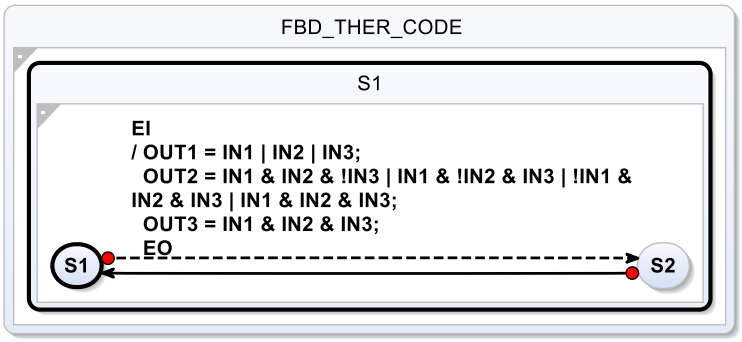


Figure H.12.: *Visualized control-flow oriented SCChart:* `FBD_THER_CODE`

```
3   output signal bool EO
4   input bool IN1
5   input bool IN2
6   input bool IN3
7   input bool IN4
8   output bool OUT1
9
10  region:
11    initial state S1{
12      initial state S1
13      immediate if EI do OUT1 = (((IN1&!(IN2)&!(IN3)&!(IN4))|(!(IN1)&IN2
            &!(IN3)&!(IN4))|(!(IN1)&!(IN2)&IN3&!(IN4))|(IN1&IN2&IN3&!(IN4)
            ))|((!(IN1)&!(IN2)&!(IN3)&IN4)|(IN1&IN2&!(IN3)&IN4)|(IN1&!(IN2
            )&IN3&IN4)|(!(IN1)&IN2&IN3&IN4))); EO abort to S2
14      final state S2
15      abort to S1
16    }
17  }
```

Listing H.13: *SCChart:* `FBD_TOGGLE_SWITCH`

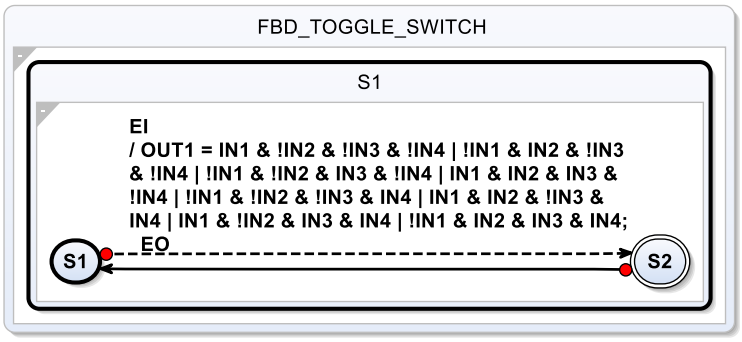


Figure H.13.: *Visualized control-flow oriented SCChart:* `FBD_TOGGLE_SWITCH`

Appendix I

ST-to-Quartz: Appendix

I.0.1. Data Types and Fields

Algorithm 37 Translate data type and field – ST-to-Quartz

Input: $\alpha^{[+]}(\omega_{st}^{\varphi})$
Output: $\alpha^{[+]}(\omega_{qrz'})$
Translation Function $t^{\alpha}_{st\mapsto qrz}(\alpha^{[+]}(\omega_{st}^{\varphi}))$**:**

$$\alpha^{[+]}(\omega_{qrz'}) \leftarrow \begin{cases} \texttt{bool} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{bv}^{bool}(\omega_{st}^{\varphi}) \\ \texttt{bv\{16\}} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{bv}^{byte}(\omega_{st}^{\varphi}) \\ \texttt{bv\{32\}} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{bv}^{word}(\omega_{st}^{\varphi}) \\ \texttt{int\{32768\}} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{i}^{int}(\omega_{st}^{\varphi}) \\ \texttt{int\{2147483648\}} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{i}^{dint}(\omega_{st}^{\varphi}) \\ \texttt{nat\{65536\}} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{i}^{uint}(\omega_{st}^{\varphi}) \\ \texttt{nat\{4294967296\}} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{i}^{udint}(\omega_{st}^{\varphi}) \\ \texttt{real} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) \in \mathcal{A}_{r}(\omega_{st}^{\varphi}) \\ \texttt{nat} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) \in \mathcal{A}_{dur}(\omega_{st}^{\varphi}) \\ \texttt{[}n(\alpha^{[+]}(\omega_{st}^{\varphi}))\texttt{+1]} \\ t^{\alpha}_{st\mapsto qrz}(\alpha^{[+]}(\omega_{st}^{\varphi})) & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) \in \mathcal{A}^{+}(\omega_{st}^{\varphi}) \end{cases}$$

I.0.2. Expressions

Algorithm 38 Translate expression – ST-to-Quartz

Input: $\tau(\omega_{st}^{\varphi})$
Output: $\tau(\omega_{qrz'})$
Translation Function $t^{\tau}_{st\mapsto qrz}(\tau(\omega_{st}^{\varphi}))$**:**

$$\tau(\omega_{qrz'}) \leftarrow \begin{cases} value(\tau(\omega_{st}^{\varphi})) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{cst}(\omega_{st}^{\varphi}) \\ id(\tau(\omega_{st}^{\varphi})) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{id}(\omega_{st}^{\varphi}) \\ (t^{\tau}_{st\mapsto qrz}(\tau(\omega_{st}^{\varphi}))) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{br}(\omega_{st}^{\varphi}) \\ \texttt{true} & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{true}(\omega_{st}^{\varphi}) \\ \texttt{false} & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{false}(\omega_{st}^{\varphi}) \\ \texttt{x[n]} & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{arr}(\omega_{st}^{\varphi}) \\ \texttt{x\_y} & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{inv}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ == } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{eq}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ != } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{ne}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ > } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{gt}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ >= } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{ge}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ < } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{lt}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ <= } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{le}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ * } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{mul}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ / } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{div}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ + } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{add}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ - } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{sub}(\omega_{st}^{\varphi}) \\ \texttt{exp}(\eta_1^r, \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{expt}(\omega_{st}^{\varphi}) \\ (\eta_1^i \texttt{ \% } \eta_2^i) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{mod}(\omega_{st}^{\varphi}) \\ \texttt{-}(\eta_1) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{um}(\omega_{st}^{\varphi}) \\ \lambda^b \texttt{ ? } \pi_1 \texttt{ : } \pi_2 & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{cond}^{sel}(\omega_{st}^{\varphi}) \\ (\lambda_1 \texttt{ \& ... \& } \lambda_n) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{bool}^{and}(\omega_{st}^{\varphi}), \alpha(\lambda) = \alpha_{bv} \\ (\lambda_1 \texttt{ | ... | } \lambda_n) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{bool}^{or}(\omega_{st}^{\varphi}), \alpha(\lambda) = \alpha_{bv} \\ (\lambda_1 \texttt{ \^{} } \lambda_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{bool}^{xor}(\omega_{st}^{\varphi}) \\ \texttt{!}(\lambda_1) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{bool}^{not}(\omega_{st}^{\varphi}), \alpha(\lambda) = \alpha_{bv} \end{cases}$$

Appendix J

ST-to-SCL: Appendix

J.0.1. Data Types and Fields

Algorithm 39 Translate data type and field – ST-to-SCL

Input: $\alpha^{[+]}(\omega_{st}^{\varphi})$
Output: $\alpha^{[+]}(\omega_{scl'})$
Translation Function $t^{\alpha}_{st\mapsto scl}(\alpha^{[+]}(\omega_{st}^{\varphi}))$**:**

$$\alpha^{[+]}(\omega_{scl'}) \leftarrow \begin{cases} \texttt{bool} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{bv}^{bool}(\omega_{st}^{\varphi}) \\ extern & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{bv}^{byte}(\omega_{st}^{\varphi}) \\ extern & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{bv}^{word}(\omega_{st}^{\varphi}) \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{int}^{int}(\omega_{st}^{\varphi}) \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{int}^{dint}(\omega_{st}^{\varphi}) \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{int}^{uint}(\omega_{st}^{\varphi}) \\ extern & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) = \alpha_{int}^{udint}(\omega_{st}^{\varphi}) \\ \texttt{float} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) \in \mathcal{A}_{r}(\omega_{st}^{\varphi}) \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) \in \mathcal{A}_{dur}(\omega_{st}^{\varphi}) \\ \begin{matrix} t^{\alpha}_{st\mapsto scl}(\alpha^{[+]}(\omega_{st}^{\varphi})) \\ \texttt{[}n(\alpha^{[+]}(\omega_{st}^{\varphi}))\texttt{+1]} \end{matrix} & \textbf{if } \alpha^{[+]}(\omega_{st}^{\varphi}) \in \mathcal{A}^{+}(\omega_{st}^{\varphi}) \end{cases}$$

J.0.2. Expressions

Algorithm 40 Translate expression – ST-to-SCL

Input: $\tau(\omega_{st}^{\varphi})$
Output: $\tau(\omega_{scl'})$
Translation Function $t_{st\mapsto scl}^{\tau}(\tau(\omega_{st}^{\varphi}))$**:**

$$\tau(\omega_{scl'}) \leftarrow \begin{cases} value(\tau(\omega_{st}^{\varphi})) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{cst}(\omega_{st}^{\varphi}) \\ id(\tau(\omega_{st}^{\varphi})) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{id}(\omega_{st}^{\varphi}) \\ t_{st\mapsto scl}^{\tau}(\tau(\omega_{st}^{\varphi})) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{\pi,\eta,\lambda}(\omega_{st}^{\varphi}) \\ (t_{st\mapsto scl}^{\tau}(\tau(\omega_{st}^{\varphi}))) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{br}(\omega_{st}^{\varphi}) \\ \texttt{true} & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{true}(\omega_{st}^{\varphi}) \\ \texttt{false} & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{false}(\omega_{st}^{\varphi}) \\ \texttt{x[n]} & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{arr}(\omega_{st}^{\varphi}) \\ extern & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{misc}^{inv}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ == } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{eq}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ != } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{ne}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ > } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{gt}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ >= } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{ge}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ < } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{lt}(\omega_{st}^{\varphi}) \\ (\pi_1 \texttt{ <= } \pi_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{comp}^{le}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ * } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{mul}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ / } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{div}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ + } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{add}(\omega_{st}^{\varphi}) \\ (\eta_1 \texttt{ - } \eta_2) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{sub}(\omega_{st}^{\varphi}) \\ extern & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{expt}(\omega_{st}^{\varphi}) \\ (\eta_1^i \texttt{ \% } \eta_2^i) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{mod}(\omega_{st}^{\varphi}) \\ \texttt{-}(\eta_1) & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{arith}^{um}(\omega_{st}^{\varphi}) \\ \lambda^b \texttt{ ? } \pi_1 \texttt{ : } \pi_2 & \textbf{if } \tau(\omega_{st}^{\varphi}) = \tau_{cond}^{sel}(\omega_{st}^{\varphi}) \\ (\lambda_1 \texttt{ \& ... \& } \lambda_n) & \textbf{if } \tau(\omega_{fbd}^{\varphi}) = \tau_{bool}^{and}(\omega_{fbd}^{\varphi}), \alpha(\lambda) = \alpha_{bv} \\ (\lambda_1 \texttt{ | ... | } \lambda_n) & \textbf{if } \tau(\omega_{fbd}^{\varphi}) = \tau_{bool}^{or}(\omega_{fbd}^{\varphi}), \alpha(\lambda) = \alpha_{bv} \\ (\lambda_1 \texttt{ ^ } \lambda_2) & \textbf{if } \tau(\omega_{fbd}^{\varphi}) = \tau_{bool}^{xor}(\omega_{fbd}^{\varphi}) \\ \texttt{!}(\lambda_1) & \textbf{if } \tau(\omega_{fbd}^{\varphi}) = \tau_{bool}^{not}(\omega_{fbd}^{\varphi}), \alpha(\lambda) = \alpha_{bv} \end{cases}$$

Appendix K

FBD-to-SCChart: Appendix

MOVE Selections Functions

```
scchart MOVE_bool {
    input bool EI
    output bool EO
    input bool IN
    output bool OUT

    region region0 {
        initial state state1
        immediate go to state4

        state state3
        go to state4

        state state4
        immediate if !EI go to state3
        immediate do OUT = IN; EO = true go to state6

        state state6
        do EO = false go to state4
    }
}
```

Listing K.1: *MOVE_bool function derived from IEC 61131-3 [GDV14]*

```
scchart MOVE_float {
    input bool EI
    output bool EO
    input float IN
    output float OUT

    region region0 {
        initial state state1
        immediate go to state4

        state state3
        go to state4

        state state4
        immediate if !EI go to state3
        immediate do OUT = IN; EO = true go to state6

        state state6
```

```
19          do EO = false go to state4
20      }
21  }
```

Listing K.2: *MOVE_float function derived from IEC 61131-3 [GDV14]*

```
1   scchart MOVE_int {
2       input bool EI
3       output bool EO
4       input int IN
5       output int OUT
6
7       region region0 {
8           initial state state1
9           immediate go to state4
10
11          state state3
12          go to state4
13
14          state state4
15          immediate if !EI go to state3
16          immediate do OUT = IN; EO = true go to state6
17
18          state state6
19          do EO = false go to state4
20      }
21  }
```

Listing K.3: *MOVE_int function derived from IEC 61131-3 [GDV14]*

Appendix L

Formal Methods-Based Optimization: Optimization Results

The optimization results include the optimization trace with the following mapping:

- 0: $\mathcal{M}_{smt}$
- 1: $f_3(f_1(\mathcal{M}_{smv}))$
- 2: $f_3(f_2(f_1(\mathcal{M}_{smv}))_{smt})$
- 3: $f_3(\mathcal{M}_{smt})$

Air Condition Control

```
# no-opt, var-opt [0, 0, 0, 0]
RS0(S⇒OR(IN1,IN2,IN3,IN4),R1⇒OR(OR(NOT(IN5),NOT(IN6),NOT(IN7),NOT(IN8))
    ,NOT(IN9)))
OUT1⇒RS0__Q1
OUT2⇒NOT(IN9)

# op-opt, edge-opt [0, 2, 0, 0]
RS0(S⇒OR(IN1,IN2,IN3,IN4),R1⇒OR(NOT(IN5),NOT(IN6),NOT(IN7),NOT(IN8),NOT
    (IN9)))
OUT1⇒RS0__Q1
OUT2⇒NOT(IN9)
```

Antivalence 3x

```
# no-opt [0]
OUT1⇒OR(AND(NOT(IN0),NOT(IN1),IN2),AND(NOT(IN0),NOT(IN2),IN1),AND(NOT(
    IN1),NOT(IN2),IN0))

# op-opt, edge-opt, var-opt [1]
OUT1⇒SEL(IN0,AND(NOT(IN1),NOT(IN2)),XOR(IN1,IN2))
```

Bending Machine Control

```
# no-opt [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
TON0(IN⇒IN1,PT⇒PT1s)
TON1(IN⇒AND(IN1,TON0__Q),PT⇒PT1s)
TON2(IN⇒IN2,PT⇒PT05s)
TOF0(IN⇒AND(IN1,TON0__Q),PT⇒PT1s)
TON3(IN⇒NOT(IN1),PT⇒PT5s)
RS0(S⇒OR(AND(IN1,TON0__Q),TON3__Q),R1⇒AND(TON1__Q,TON2__Q))
OUT1⇒AND(AND(IN1,TON0__Q),NOT(AND(TON1__Q,TON2__Q)),NOT(AND(AND(NOT(
    TON2__Q),TOF0__Q),IN3)))
OUT2⇒AND(NOT(AND(AND(NOT(TON2__Q),TOF0__Q),IN3)),AND(NOT(TON2__Q),
    TOF0__Q))
OUT3⇒RS0__Q1

# op-opt, edge-opt [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 2, 0]
TON0(IN⇒IN1,PT⇒PT1s)
TON1(IN⇒AND(IN1,TON0__Q),PT⇒PT1s)
TON2(IN⇒IN2,PT⇒PT05s)
TOF0(IN⇒AND(IN1,TON0__Q),PT⇒PT1s)
TON3(IN⇒NOT(IN1),PT⇒PT5s)
RS0(S⇒OR(AND(IN1,TON0__Q),TON3__Q),R1⇒AND(TON1__Q,TON2__Q))
OUT1⇒AND(IN1,TON0__Q,NOT(AND(TON1__Q,TON2__Q)),NOT(AND(NOT(TON2__Q),
    TOF0__Q,IN3)))
OUT2⇒AND(NOT(IN3),NOT(TON2__Q),TOF0__Q)
OUT3⇒RS0__Q1

# var-opt [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
TON0(IN⇒IN1,PT⇒PT1s)
TON1(IN⇒AND(IN1,TON0__Q),PT⇒PT1s)
TON2(IN⇒IN2,PT⇒PT05s)
TOF0(IN⇒AND(IN1,TON0__Q),PT⇒PT1s)
TON3(IN⇒NOT(IN1),PT⇒PT5s)
RS0(S⇒OR(AND(IN1,TON0__Q),TON3__Q),R1⇒AND(TON1__Q,TON2__Q))
OUT1⇒AND(AND(IN1,TON0__Q),NOT(AND(TON1__Q,TON2__Q)),NOT(AND(AND(NOT(
    TON2__Q),TOF0__Q),IN3)))
OUT2⇒AND(AND(NOT(IN3),NOT(TON2__Q)),TOF0__Q)
OUT3⇒RS0__Q1
```

Cylinder Control System

```
# no-opt, var-opt [0, 0, 0, 0, 0, 0]
OUTBp⇒AND(INa1,AND(OR(ME1,AND(INa0,INS)),NOT(INb1)))
OUTAp⇒AND(OR(ME1,AND(INa0,INS)),NOT(INb1))
OUTAm⇒AND(NOT(AND(OR(ME1,AND(INa0,INS)),NOT(INb1))),INc0,NOT(AND(OR(ME2
    ,INb1),NOT(INc1))))
OUTCp⇒AND(AND(OR(ME2,INb1),NOT(INc1)),INb0)
ME2⇒AND(OR(ME2,INb1),NOT(INc1))
ME1⇒AND(OR(ME1,AND(INa0,INS)),NOT(INb1))

# op-opt
[2, 0, 2, 2, 0, 0]
OUTBp⇒AND(INa1,NOT(INb1),SEL(INa0,OR(INS,ME1),ME1))
OUTAp⇒AND(OR(ME1,AND(INa0,INS)),NOT(INb1))
OUTAm⇒AND(INc0,SEL(AND(INa0,INS),AND(INb1,INc1),SEL(INb1,INc1,AND(NOT(
    ME1),OR(INc1,NOT(ME2))))))
OUTCp⇒AND(NOT(INc1),INb0,OR(INb1,ME2))
ME2⇒AND(OR(ME2,INb1),NOT(INc1))
ME1⇒AND(OR(ME1,AND(INa0,INS)),NOT(INb1))

# edge-opt [3, 0, 0, 2, 0, 0]
OUTBp⇒AND(INa1,OR(ME1,AND(INa0,INS)),NOT(INb1))
```

```
OUTAp⇒AND(OR(ME1,AND(INa0,INS)),NOT(INb1))
OUTAm⇒AND(NOT(AND(OR(ME1,AND(INa0,INS)),NOT(INb1))),INc0,NOT(AND(OR(ME2
    ,INb1),NOT(INc1))))
OUTCp⇒AND(NOT(INc1),INb0,OR(INb1,ME2))
ME2⇒AND(OR(ME2,INb1),NOT(INc1))
ME1⇒AND(OR(ME1,AND(INa0,INS)),NOT(INb1))
```

Dice Numbers Indicator

```
# no-opt [0, 0, 0, 0, 0, 0, 0, 0, 0]
OUTa⇒OR(AND(IN3,NOT(IN4),NOT(IN2)),AND(IN2,NOT(IN4),NOT(IN1)),AND(IN4,
    NOT(IN2),NOT(IN3)))
OUTb⇒OR(AND(IN2,IN3,NOT(IN4)),AND(IN4,NOT(IN2),NOT(IN3)))
OUTc⇒OR(AND(IN3,NOT(IN4)),AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,NOT(
    IN4)))
OUTd⇒OR(AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,IN3,NOT(IN4)))
OUTe⇒OR(AND(IN1,NOT(IN4)),AND(IN1,NOT(IN3),NOT(IN2)))
OUTf⇒OR(AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,IN3,NOT(IN4)))
OUTh⇒OR(AND(IN2,IN3,NOT(IN4)),AND(IN4,NOT(IN2),NOT(IN3)))
OUTg⇒OR(AND(IN3,NOT(IN4)),AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,NOT(
    IN4)))
OUTi⇒OR(AND(IN3,NOT(IN4),NOT(IN2)),AND(IN2,NOT(IN4),NOT(IN1),NOT(IN3)),
    AND(IN4,NOT(IN2),NOT(IN3)))

# op-opt [1, 1, 1, 0, 0, 0, 1, 1, 1]
OUTa⇒SEL(IN1,AND(NOT(IN2),XOR(IN3,IN4)),XOR(OR(IN2,IN3),IN4))
OUTb⇒SEL(IN2,AND(IN3,NOT(IN4)),AND(NOT(IN3),IN4))
OUTc⇒SEL(IN1,XOR(OR(IN2,IN3),IN4),SEL(IN2,AND(IN3,NOT(IN4)),XOR(IN3,IN4
    )))
OUTd⇒OR(AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,IN3,NOT(IN4)))
OUTe⇒OR(AND(IN1,NOT(IN4)),AND(IN1,NOT(IN3),NOT(IN2)))
OUTf⇒OR(AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,IN3,NOT(IN4)))
OUTh⇒SEL(IN2,AND(IN3,NOT(IN4)),AND(NOT(IN3),IN4))
OUTg⇒SEL(IN1,XOR(OR(IN2,IN3),IN4),SEL(IN2,AND(IN3,NOT(IN4)),XOR(IN3,IN4
    )))
OUTi⇒SEL(IN1,AND(NOT(IN2),XOR(IN3,IN4)),SEL(IN2,AND(NOT(IN3),NOT(IN4)),
    XOR(IN3,IN4)))

# edge-opt, var-opt [1, 1, 0, 0, 0, 0, 1, 0, 1]
OUTa⇒SEL(IN1,AND(NOT(IN2),XOR(IN3,IN4)),XOR(OR(IN2,IN3),IN4))
OUTb⇒SEL(IN2,AND(IN3,NOT(IN4)),AND(NOT(IN3),IN4))
OUTc⇒OR(AND(IN3,NOT(IN4)),AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,NOT(
    IN4)))
OUTd⇒OR(AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,IN3,NOT(IN4)))
OUTe⇒OR(AND(IN1,NOT(IN4)),AND(IN1,NOT(IN3),NOT(IN2)))
OUTf⇒OR(AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,IN3,NOT(IN4)))
OUTh⇒SEL(IN2,AND(IN3,NOT(IN4)),AND(NOT(IN3),IN4))
OUTg⇒OR(AND(IN3,NOT(IN4)),AND(IN4,NOT(IN2),NOT(IN3)),AND(IN1,IN2,NOT(
    IN4)))
OUTi⇒SEL(IN1,AND(NOT(IN2),XOR(IN3,IN4)),SEL(IN2,AND(NOT(IN3),NOT(IN4)),
    XOR(IN3,IN4)))
```

KV Diagram optimized Chart

```
# no-opt [0, 0]
OUT1⇒OR(AND(NOT(INa),INb),AND(INa,NOT(INb),INc))
OUT2⇒AND(OR(AND(EQ(INa,INb),INc),AND(NOT(INa),NOT(INc),INb)),NOT(INd))

# op-opt, edge-opt, var-opt [1, 1]
OUT1⇒SEL(INa,AND(NOT(INb),INc),INb)
OUT2⇒AND(NOT(INd),SEL(INa,AND(INb,INc),XOR(INb,INc)))
```

Pollutant Indicator

```
# no-opt [0, 0, 0]
OUT1⇒OR(AND(NOT(IN3),NOT(IN2),IN1),AND(NOT(IN3),IN2,NOT(IN1)),AND(IN3,
    NOT(IN2),NOT(IN1)))
OUT2⇒OR(AND(NOT(IN3),IN2,IN1),AND(IN3,NOT(IN2),IN1),AND(IN3,IN2,NOT(IN1
    )))
OUT3⇒AND(IN1,IN2,IN3)

# op-opt, edge-opt, var-opt [1, 1, 0]
OUT1⇒SEL(IN1,AND(NOT(IN2),NOT(IN3)),XOR(IN2,IN3))
OUT2⇒SEL(IN1,XOR(IN2,IN3),AND(IN2,IN3))
OUT3⇒AND(IN1,IN2,IN3)
```

Reservoirs Control System 1

```
# no-opt [0, 0, 0, 0]
OUTP1⇒OR(AND(NOT(IN4),NOT(IN3)),AND(NOT(IN4),NOT(IN2),IN1))
OUTP2⇒OR(OR(AND(NOT(IN4),NOT(IN3)),AND(NOT(IN4),NOT(IN2),IN1)),AND(NOT(
    IN3),IN2,NOT(IN1)))
OUTP3⇒OR(AND(NOT(IN4),NOT(IN3),NOT(IN2)),AND(IN4,NOT(IN3),IN2),AND(NOT(
    IN3),IN2,NOT(IN1)),AND(NOT(IN4),IN3,IN2,IN2))
OUTH⇒OR(AND(IN3,NOT(IN1)),AND(IN4,NOT(IN2)))

# op-opt [0, 1, 0, 0]
OUTP1⇒OR(AND(NOT(IN4),NOT(IN3)),AND(NOT(IN4),NOT(IN2),IN1))
OUTP2⇒SEL(IN1,AND(NOT(IN4),OR(NOT(IN2),NOT(IN3))),AND(NOT(IN3),OR(IN2,
    NOT(IN4))))
OUTP3⇒OR(AND(NOT(IN4),NOT(IN3),NOT(IN2)),AND(IN4,NOT(IN3),IN2),AND(NOT(
    IN3),IN2,NOT(IN1)),AND(NOT(IN4),IN3,IN2,IN2))
OUTH⇒OR(AND(IN3,NOT(IN1)),AND(IN4,NOT(IN2)))

# edge-opt, var-opt [0, 1, 1, 0]
OUTP1⇒OR(AND(NOT(IN4),NOT(IN3)),AND(NOT(IN4),NOT(IN2),IN1))
OUTP2⇒SEL(IN1,AND(NOT(IN4),OR(NOT(IN2),NOT(IN3))),AND(NOT(IN3),OR(IN2,
    NOT(IN4))))
OUTP3⇒SEL(IN1,SEL(IN2,XOR(IN3,IN4),AND(NOT(IN3),NOT(IN4))),SEL(IN2,OR(
    NOT(IN3),NOT(IN4)),AND(NOT(IN3),NOT(IN4))))
OUTH⇒OR(AND(IN3,NOT(IN1)),AND(IN4,NOT(IN2)))
```

Reservoirs Control System 2

```
# no-opt [0, 0, 0, 0]
OUTP3⇒OR(AND(NOT(IN4),NOT(IN3)),AND(NOT(IN4),NOT(IN2),IN1))
OUTP2⇒OR(OR(AND(NOT(IN4),NOT(IN3)),AND(NOT(IN4),NOT(IN2),IN1)),AND(NOT(
    IN3),IN2,NOT(IN1)))
OUTP1⇒OR(AND(NOT(IN4),NOT(IN3),NOT(IN2)),AND(IN4,NOT(IN3),IN2,IN1),AND(
    NOT(IN3),IN2,NOT(IN1)),AND(NOT(IN4),IN3,IN2,IN1))
OUTQ⇒OR(AND(IN3,NOT(IN1)),AND(IN4,NOT(IN2)))

# op-opt, edge-opt, var-opt [0, 1, 1, 0]
OUTP3⇒OR(AND(NOT(IN4),NOT(IN3)),AND(NOT(IN4),NOT(IN2),IN1))
OUTP2⇒SEL(IN1,AND(NOT(IN4),OR(NOT(IN2),NOT(IN3))),AND(NOT(IN3),OR(IN2,
    NOT(IN4))))
OUTP1⇒SEL(IN1,SEL(IN2,XOR(IN3,IN4),AND(NOT(IN3),NOT(IN4))),AND(NOT(IN3)
    ,OR(IN2,NOT(IN4))))
OUTQ⇒OR(AND(IN3,NOT(IN1)),AND(IN4,NOT(IN2)))
```

Roll Down Shutters

```
# no-opt [0, 0]
OUT1⇒AND(OR(AND(TIME4,IN6),AND(NOT(IN6),IN2)),IN4)
OUT2⇒AND(EQ(AND(OR(AND(TIME4,IN6),AND(NOT(IN6),IN2)),IN4),AND(IN5,OR(
    AND(IN6,IN1,NOT(TIME14)),AND(IN3,NOT(IN6))))),AND(IN5,OR(AND(IN6,IN1
    ,NOT(TIME14)),AND(IN3,NOT(IN6)))))

# op-opt, edge-opt, var-opt [0, 2]
OUT1⇒AND(OR(AND(TIME4,IN6),AND(NOT(IN6),IN2)),IN4)
OUT2⇒AND(IN4,IN5,SEL(IN1,SEL(AND(IN2,IN3),OR(NOT(IN6),AND(NOT(TIME14),
    TIME4)),AND(IN6,NOT(TIME14),TIME4)),AND(IN2,IN3,NOT(IN6))))
```

Cable winch

```
# no-opt [0, 0, 0, 0]
OUT1⇒AND(OR(ME1,INS1,NOT(INB2)),NOT(OR(NOT(INB1),NOT(INS1),ME2)))
OUT2⇒AND(OR(ME2,NOT(INB1)),NOT(OR(NOT(INB2),NOT(INS1),ME1)))
ME1⇒AND(OR(ME1,INS1,NOT(INB2)),NOT(OR(NOT(INB1),NOT(INS1),ME2)))
ME2⇒AND(OR(ME2,NOT(INB1)),NOT(OR(NOT(INB2),NOT(INS1),ME1)))

# op-opt, edge-opt [2, 2, 2, 2]
OUT1⇒AND(INS1,INB1,NOT(ME2))
OUT2⇒AND(INS1,INB2,NOT(ME1),OR(ME2,NOT(INB1)))
ME1⇒AND(INS1,INB1,NOT(ME2))
ME2⇒AND(INS1,INB2,NOT(ME1),OR(ME2,NOT(INB1)))

# var-opt [1, 0, 1, 0]
OUT1⇒AND(AND(INS1,INB1),NOT(ME2))
OUT2⇒AND(OR(ME2,NOT(INB1)),NOT(OR(NOT(INB2),NOT(INS1),ME1)))
ME1⇒AND(AND(INS1,INB1),NOT(ME2))
ME2⇒AND(OR(ME2,NOT(INB1)),NOT(OR(NOT(INB2),NOT(INS1),ME1)))
```

Seven Segment Display

```
# no-opt [0, 0, 0, 0, 0, 0, 0]
OUTa⇒OR(AND(NOT(IN3),IN1),AND(IN2,IN1),AND(NOT(IN2),NOT(IN1),IN3),AND(
    NOT(IN3),NOT(IN2),NOT(IN0)),AND(NOT(IN3),IN1,IN0))
OUTb⇒OR(AND(NOT(IN3),NOT(IN2)),AND(NOT(IN2),NOT(IN1)),AND(NOT(IN3),IN1,
    IN0),AND(IN3,NOT(IN1),IN0),AND(NOT(IN3),NOT(IN1),NOT(IN0)))
OUTc⇒OR(AND(NOT(IN3),NOT(IN1)),AND(NOT(IN3),IN0),AND(NOT(IN2),IN3),AND(
    NOT(IN3),IN2),AND(NOT(IN1),IN0))
OUTd⇒OR(AND(NOT(IN2),IN1),AND(NOT(IN2),NOT(IN0)),AND(IN1,NOT(IN0)),AND(
    IN3,NOT(IN1),IN2),AND(IN2,NOT(IN1),IN0))
OUTe⇒OR(AND(NOT(IN2),NOT(IN0)),AND(IN3,IN1),AND(IN3,IN2),AND(IN2,NOT(
    IN0),IN1))
OUTf⇒OR(AND(IN3,IN2,IN1),AND(NOT(IN3),IN2,NOT(IN1)),AND(NOT(IN3),IN2,
    NOT(IN0)),AND(IN3,NOT(IN2),IN0),AND(NOT(IN2),NOT(IN1),NOT(IN0)))
OUTg⇒OR(IN3,AND(NOT(IN0),IN2),AND(NOT(IN2),IN0),AND(NOT(IN1),IN2))

# op-opt, edge-opt [1, 1, 2, 2, 2, 1, 2]
OUTa⇒SEL(IN0,SEL(IN1,OR(IN2,NOT(IN3)),AND(NOT(IN2),IN3)),SEL(IN1,OR(IN2
    ,NOT(IN3)),NOT(IN2)))
OUTb⇒SEL(IN0,SEL(IN1,NOT(IN3),OR(NOT(IN2),IN3)),SEL(IN1,AND(NOT(IN2),
    NOT(IN3)),OR(NOT(IN2),NOT(IN3))))
OUTc⇒SEL(IN0,OR(NOT(IN3),NOT(IN2),NOT(IN1)),SEL(IN1,XOR(IN3,IN2),OR(NOT
    (IN2),NOT(IN3))))
OUTd⇒SEL(IN0,XOR(IN2,IN1),OR(IN1,IN3,NOT(IN2)))
OUTe⇒SEL(IN0,AND(IN3,OR(IN1,IN2)),OR(IN1,IN3,NOT(IN2)))
```

```
OUTf⇒SEL(IN0,SEL(IN1,IN3,XOR(IN2,IN3)),SEL(IN1,IN2,OR(NOT(IN2),NOT(IN3)
    )))
OUTg⇒SEL(IN0,OR(IN3,NOT(IN2),NOT(IN1)),OR(IN2,IN3))

# var-opt [1, 1, 1, 1, 1, 1, 1]
OUTa⇒SEL(IN0,SEL(IN1,OR(IN2,NOT(IN3)),AND(NOT(IN2),IN3)),SEL(IN1,OR(IN2
    ,NOT(IN3)),NOT(IN2)))
OUTb⇒SEL(IN0,SEL(IN1,NOT(IN3),OR(NOT(IN2),IN3)),SEL(IN1,AND(NOT(IN2),
    NOT(IN3)),OR(NOT(IN2),NOT(IN3))))
OUTc⇒SEL(IN0,OR(NOT(IN1),OR(NOT(IN2),NOT(IN3))),SEL(IN1,XOR(IN2,IN3),OR
    (NOT(IN2),NOT(IN3))))
OUTd⇒SEL(IN0,XOR(IN1,IN2),OR(IN1,OR(NOT(IN2),IN3)))
OUTe⇒SEL(IN0,AND(IN3,OR(IN1,IN2)),OR(IN1,OR(NOT(IN2),IN3)))
OUTf⇒SEL(IN0,SEL(IN1,IN3,XOR(IN2,IN3)),SEL(IN1,IN2,OR(NOT(IN2),NOT(IN3)
    )))
OUTg⇒SEL(IN0,OR(NOT(IN1),OR(NOT(IN2),IN3)),OR(IN2,IN3))
```

Shop Window Lighting

```
# no-opt [0, 0, 0, 0, 0, 0, 0, 0]
TOF0(IN⇒IN3,PT⇒PT1m)
TOF1(IN⇒IN4,PT⇒PT1m)
OUT1⇒AND(OR(TIME3,TOF0__Q),IN2)
OUT2⇒AND(IN2,OR(AND(AND(OR(TIME3,TOF0__Q),IN2),IN1),TOF0__Q))
OUT3⇒AND(IN2,OR(AND(IN1,NOT(AND(OR(TIME3,TOF0__Q),IN2))),TOF0__Q))
OUT4⇒AND(IN2,OR(TOF0__Q,TOF1__Q))

# op-opt, edge-opt, var-opt [0, 0, 0, 0, 0, 1, 1, 0]
TOF0(IN⇒IN3,PT⇒PT1m)
TOF1(IN⇒IN4,PT⇒PT1m)
OUT1⇒AND(OR(TIME3,TOF0__Q),IN2)
OUT2⇒AND(IN2,SEL(IN1,OR(TIME3,TOF0__Q),TOF0__Q))
OUT3⇒AND(IN2,SEL(IN1,OR(NOT(TIME3),TOF0__Q),TOF0__Q))
OUT4⇒AND(IN2,OR(TOF0__Q,TOF1__Q))
```

Silo Valve Control System

```
# no-opt [0]
OUT1⇒OR(AND(NOT(IN3),NOT(IN2),IN1),AND(NOT(IN3),IN2,NOT(IN1)),AND(IN3,
    NOT(IN2),NOT(IN1)),AND(IN3,IN2,IN1))

# op-opt, edge-opt, var-opt [2]
OUT1⇒SEL(IN1,EQ(IN2,IN3),XOR(IN3,IN2))
```

Smoke Detection System

```
# no-opt [0, 0, 0]
OUT1⇒OR(NOT(IN1),NOT(IN2),NOT(IN3),NOT(IN4))
OUT2⇒OR(OR(AND(NOT(IN1),NOT(IN2)),AND(NOT(IN1),NOT(IN3))),AND(NOT(IN1),
    NOT(IN4),AND(NOT(IN3),NOT(IN2)),AND(NOT(IN2),NOT(IN4)),AND(NOT(IN3),
    NOT(IN4))))
OUT3⇒AND(NOT(IN1),NOT(IN2),NOT(IN3),NOT(IN4))

# op-opt, edge-opt, var-opt [0, 1, 0]
OUT1⇒OR(NOT(IN1),NOT(IN2),NOT(IN3),NOT(IN4))
OUT2⇒AND(NOT(IN1),OR(NOT(IN2),NOT(IN3)))
OUT3⇒AND(NOT(IN1),NOT(IN2),NOT(IN3),NOT(IN4))
```

Sports Hall Lighting

```
# no-opt, var-opt [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
RS0(S⇒IN1,R1⇒TIME4)
RS1(S⇒IN1,R1⇒TIME9)
RS2(S⇒IN2,R1⇒TIME14)
TON0(IN⇒TIME18,PT⇒PT1m)
OUT1⇒OR(AND(RS0__Q1,NOT(IN4)),IN3)
OUT2⇒OR(AND(RS1__Q1,NOT(IN4)),IN3)
OUT3⇒OR(AND(RS2__Q1,NOT(IN4)),IN3)
OUT4⇒AND(EQ(TIME18,TON0__Q),OR(AND(RS0__Q1,NOT(IN4)),IN3),OR(AND(
    RS1__Q1,NOT(IN4)),IN3))

# op-opt, edge-opt [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2]
RS0(S⇒IN1,R1⇒TIME4)
RS1(S⇒IN1,R1⇒TIME9)
RS2(S⇒IN2,R1⇒TIME14)
TON0(IN⇒TIME18,PT⇒PT1m)
OUT1⇒OR(AND(RS0__Q1,NOT(IN4)),IN3)
OUT2⇒OR(AND(RS1__Q1,NOT(IN4)),IN3)
OUT3⇒OR(AND(RS2__Q1,NOT(IN4)),IN3)
OUT4⇒SEL(IN3,EQ(TIME18,TON0__Q),AND(NOT(IN4),RS0__Q1,RS1__Q1,EQ(TIME18,
    TON0__Q)))
```

Thermometer Code System

```
# no-opt [0, 0, 0]
OUT1⇒OR(IN1,IN2,IN3)
OUT2⇒OR(AND(IN1,IN2,NOT(IN3)),AND(IN1,NOT(IN2),IN3),AND(NOT(IN1),IN2,
    IN3),AND(IN1,IN2,IN3))
OUT3⇒AND(IN1,IN2,IN3)

# op-opt, edge-opt, var-opt [0, 1, 0]
OUT1⇒OR(IN1,IN2,IN3)
OUT2⇒SEL(IN1,OR(IN2,IN3),AND(IN2,IN3))
OUT3⇒AND(IN1,IN2,IN3)
```

Toggle Switch 4x

```
# no-opt [0]
OUT1⇒OR(OR(AND(IN1,NOT(IN2),NOT(IN3),NOT(IN4)),AND(NOT(IN1),IN2,NOT(IN3
    ),NOT(IN4)),AND(NOT(IN1),NOT(IN2),IN3,NOT(IN4)),AND(IN1,IN2,IN3,NOT(
    IN4))),OR(AND(NOT(IN1),NOT(IN2),NOT(IN3),IN4),AND(IN1,IN2,NOT(IN3),
    IN4),AND(IN1,NOT(IN2),IN3,IN4),AND(NOT(IN1),IN2,IN3,IN4)))

# op-opt, edge-opt, var-opt [2]
OUT1⇒SEL(OR(AND(IN1,IN2),AND(NOT(IN1),NOT(IN2))),XOR(IN4,IN3),EQ(IN3,
    IN4))
```

Ventilation Control System

```
# no-opt [0, 0, 0]
OUT1⇒AND(NOT(IN1),IN2,IN3,IN4)
OUT2⇒OR(AND(IN2,NOT(IN4),NOT(IN1)),AND(IN4,NOT(IN3),NOT(IN1)),AND(NOT(
    IN3),IN2,NOT(IN4)),AND(NOT(IN1),IN3,NOT(IN2)),AND(NOT(IN4),IN1,NOT(
    IN2)),AND(NOT(IN3),IN1,NOT(IN2)))
```

```
OUT3⇒OR(AND(IN1,IN3,IN4),AND(IN1,IN2,IN3),AND(IN1,IN2,IN4))

# op-opt [0, 1, 0]
OUT1⇒AND(NOT(IN1),IN2,IN3,IN4)
OUT2⇒SEL(IN1,SEL(IN2,AND(NOT(IN3),NOT(IN4)),OR(NOT(IN3),NOT(IN4))),SEL(
    IN2,OR(NOT(IN3),NOT(IN4)),OR(IN3,IN4)))
OUT3⇒OR(AND(IN1,IN3,IN4),AND(IN1,IN2,IN3),AND(IN1,IN2,IN4))

# edge-opt, var-opt [0, 1, 1]
OUT1⇒AND(NOT(IN1),IN2,IN3,IN4)
OUT2⇒SEL(IN1,SEL(IN2,AND(NOT(IN3),NOT(IN4)),OR(NOT(IN3),NOT(IN4))),SEL(
    IN2,OR(NOT(IN3),NOT(IN4)),OR(IN3,IN4)))
OUT3⇒AND(IN1,SEL(IN2,OR(IN3,IN4),AND(IN3,IN4)))
```

Wind Direction Indicator

```
# no-opt [0, 0, 0, 0]
OUT1⇒OR(AND(IN1,NOT(IN2),NOT(IN3)),AND(NOT(IN1),NOT(IN2),NOT(IN3)),AND(
    NOT(IN1),NOT(IN2),IN3))
OUT2⇒OR(AND(NOT(IN1),NOT(IN2),IN3),AND(IN1,NOT(IN2),IN3),AND(IN1,IN2,
    IN3))
OUT3⇒OR(AND(IN1,IN2,IN3),AND(NOT(IN1),IN2,IN3),AND(NOT(IN1),IN2,NOT(IN3
    )))
OUT4⇒OR(AND(NOT(IN1),IN2,NOT(IN3)),AND(IN1,IN2,NOT(IN3)),AND(IN1,NOT(
    IN2),NOT(IN3)))

# op-opt, edge-opt, var-opt [1, 1, 1, 1]
OUT1⇒AND(NOT(IN2),OR(NOT(IN1),NOT(IN3)))
OUT2⇒AND(IN3,OR(IN1,NOT(IN2)))
OUT3⇒AND(IN2,OR(NOT(IN1),IN3))
OUT4⇒AND(NOT(IN3),OR(IN1,IN2))
```

Appendix M

Quartz-to-SCChart: Appendix

M.0.1. Data Types and Fields

Algorithm 41 Translate data type and field – Quartz-to-SCChart

Input: $\alpha^{[+]}(\omega_{qrz})$
Output: $\alpha^{[+]}(\omega_{scc'})$
Translation Function $t^{\alpha}_{qrz\mapsto scc}(\alpha^{[+]}(\omega_{qrz}))$**:**

$$\alpha^{[+]}(\omega_{scc'}) \leftarrow \begin{cases} \texttt{bool} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) = \alpha^{bool}_{bv}(\omega_{qrz}) \\ \texttt{extern} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) = \alpha^{byte}_{bv}(\omega_{qrz}) \\ \texttt{extern} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) = \alpha^{word}_{bv}(\omega_{qrz}) \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) = \texttt{int}\{32768\} \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) = \texttt{int}\{2147483648\} \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) = \texttt{nat}\{65536\} \\ \texttt{extern} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) = \texttt{nat}\{4294967296\} \\ \texttt{float} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) \in \mathcal{A}_r(\omega_{qrz}) \\ \texttt{int} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) \in \mathcal{A}_{dur}(\omega_{qrz}) \\ \begin{matrix}\texttt{[}n(\alpha^{[+]}(\omega_{qrz}))\texttt{+1]} \\ \mathrm{t}^{\alpha}_{qrz\mapsto scc}(\alpha^{[+]}(\omega_{qrz}))\end{matrix} & \textbf{if } \alpha^{[+]}(\omega_{qrz}) \in \mathcal{A}^{+}(\omega_{qrz}) \end{cases}$$

M.0.2. Expressions

Algorithm 42 Translate expression – Quartz-to-SCChart

Input: $\tau(\omega_{qrz})$
Output: $\tau(\omega_{scc'})$
Translation Function $t^{\tau}_{qrz \mapsto scc}(\tau(\omega_{qrz}))$**:**

$$
\tau(\omega_{scc'}) \leftarrow \begin{cases}
value(\tau(\omega_{qrz})) & \textbf{if } \texttt{emit()} \\
value(\tau(\omega_{qrz})) & \textbf{if } \tau(\omega_{qrz}) = \tau^{cst}_{misc}(\omega_{qrz}) \\
id(\tau(\omega_{qrz})) & \textbf{if } \tau(\omega_{qrz}) = \tau^{id}_{misc}(\omega_{qrz}) \\
t^{\tau}_{qrz \mapsto scc}(\tau(\omega_{qrz})) & \textbf{if } \tau(\omega_{qrz}) = \tau^{\pi,\eta,\lambda}_{misc}(\omega_{qrz}) \\
(t^{\tau}_{qrz \mapsto scc}(\tau(\omega_{qrz}))) & \textbf{if } \tau(\omega_{qrz}) = \tau^{br}_{misc}(\omega_{qrz}) \\
\texttt{true} & \textbf{if } \tau(\omega_{qrz}) = \tau^{true}_{misc}(\omega_{qrz}) \\
\texttt{false} & \textbf{if } \tau(\omega_{qrz}) = \tau^{false}_{misc}(\omega_{qrz}) \\
\texttt{x[n]} & \textbf{if } \tau(\omega_{qrz}) = \tau^{arr}_{misc}(\omega_{qrz}) \\
extern & \textbf{if } \tau(\omega_{qrz}) = \tau^{inv}_{misc}(\omega_{qrz}) \\
(\pi_1 \texttt{ == } \pi_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{eq}_{comp}(\omega_{qrz}) \\
(\pi_1 \texttt{ != } \pi_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{ne}_{comp}(\omega_{qrz}) \\
(\pi_1 \texttt{ > } \pi_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{gt}_{comp}(\omega_{qrz}) \\
(\pi_1 \texttt{ >= } \pi_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{ge}_{comp}(\omega_{qrz}) \\
(\pi_1 \texttt{ < } \pi_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{lt}_{comp}(\omega_{qrz}) \\
(\pi_1 \texttt{ <= } \pi_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{le}_{comp}(\omega_{qrz}) \\
(\eta_1 \texttt{ * } \eta_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{mul}_{arith}(\omega_{qrz}) \\
(\eta_1 \texttt{ / } \eta_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{div}_{arith}(\omega_{qrz}) \\
(\eta_1 \texttt{ + } \eta_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{add}_{arith}(\omega_{qrz}) \\
(\eta_1 \texttt{ - } \eta_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{sub}_{arith}(\omega_{qrz}) \\
extern & \textbf{if } \tau(\omega_{qrz}) = \tau^{expt}_{arith}(\omega_{qrz}) \\
(\eta^i_1 \texttt{ \% } \eta^i_2) & \textbf{if } \tau(\omega_{qrz}) = \tau^{mod}_{arith}(\omega_{qrz}) \\
\texttt{-}(\eta_1) & \textbf{if } \tau(\omega_{qrz}) = \tau^{um}_{arith}(\omega_{qrz}) \\
\lambda^b \texttt{ ? } \pi_1 \texttt{ : } \pi_2 & \textbf{if } \tau(\omega_{qrz}) = \tau^{sel}_{cond}(\omega_{qrz}) \\
(\lambda_1 \texttt{ \& ... \& } \lambda_n) & \textbf{if } \tau(\omega^{\varphi}_{fbd}) = \tau^{and}_{bool}(\omega^{\varphi}_{fbd}) \\
(\lambda_1 \texttt{ | ... | } \lambda_n) & \textbf{if } \tau(\omega^{\varphi}_{fbd}) = \tau^{or}_{bool}(\omega^{\varphi}_{fbd}) \\
(\lambda_1 \texttt{ \^{} } \lambda_2) & \textbf{if } \tau(\omega^{\varphi}_{fbd}) = \tau^{xor}_{bool}(\omega^{\varphi}_{fbd}) \\
\texttt{!}(\lambda_1) & \textbf{if } \tau(\omega^{\varphi}_{fbd}) = \tau^{not}_{bool}(\omega^{\varphi}_{fbd}), \alpha(\lambda) = \alpha_{bv}
\end{cases}
$$

Appendix N

Curriculum Vitae

Berufserfahrung

seit 2023	**Software & Control Engineer Team Leader** ALSTOM Transportation Germany GmbH, Mannheim
seit 2019	**Lehrbeauftragter** Duale Hochschule Baden-Württemberg, Mannheim
2017–2023	**Software Development & Test Engineer,** **Software Verification & Test Leader** ALSTOM Transportation Germany GmbH (ehemals Bombardier Transportation GmbH), Mannheim
2015–2017	**Projektingenieur** Vision Machine Technic Bildverarbeitungssysteme GmbH, Mannheim
2011–2012	**Energieberater** Technische Werke Ludwigshafen AG, Ludwigshafen am Rhein

Akademische Ausbildung

2016–2019	**Master of Science, Praktische Informatik (Note 1,4)** FernUniversität in Hagen, Hagen Masterarbeit: *Reengineering von IEC 61131-3-basierten Applikationslösungen*
2012–2015	**Bachelor of Engineering, Elektrotechnik (Note 2,1)** Duale Hochschule Baden-Württemberg, Mannheim | ABB Training Center GmbH & Co. KG, Heidelberg Bachelorarbeit: *Modeling and Hardware-in-the-Loop Simulation of Automation Solutions*

Berufliche Ausbildung

2007–2011	**Elektroniker für Betriebstechnik** Technische Werke Ludwigshafen AG, Ludwigshafen am Rhein

Schulausbildung

2009–2010	**Fachhochschulreife** Berufsbildende Schule Technik I, Ludwigshafen am Rhein
2001–2007	**Mittlere Reife** Wilhelm-von-Humboldt Gymnasium, Ludwigshafen am Rhein