Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Code Generation for Sequential Constructiveness

Steven Patrick Smyth

July 24, 2013

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl.-Inf. Christian Motika

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Abstract

Many programming languages in the synchronous world employ a classical Model of Computation which allows them to implement determinism since they rule out race conditions. However, to do so, they impose heavy restrictions on which programs are considered valid. The newly refined Sequentially Constructive Model of Computation, developed by von Hanxleden et al. in 2012, aims to lift some of these restrictions by allowing sequential and concurrent dependent variable accesses to proceed as long as the program stays statically schedulable.

The code generation approach presented in this thesis introduces a chain of key steps for deriving code automatically out of sequentially constructive statecharts, or SCCharts. It explains the intermediate language SCL, its graphical representation, the SCG, and clarifies each particular step of the transformation chain. Especially the determination of schedulability is elucidated. Additionally, pointers to optimizations are given and since the approach is embedded in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), experimental results are directly compared to concepts already present in KIELER.

Key words synchronous languages, modelling languages, constructiveness, sequential constructiveness, SCCharts, SCL, SCG, SyncCharts, KIELER, code generation, schedulability

Contents

List of Figures

Listings

Abbreviations

ANDEXP	Logical AND Expression
ASSEXP	Assignment expression
ASC	Acyclic Sequentially Constructive
API	Application Programming Interface
BB	Basic Block
BBP	Basic Block Predecessor
BBS	Basic Block Successor
DSL	Domain Specific Language
DTO	Duplicate Transition Optimization
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
FSM	Finite State Machine
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
ID	Identifier
IDE	Integrated Development Environment
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager
KIML	KIELER Infrastructure for Meta Layout
KIVi	KIELER View Management
KLay	KIELER Layouters
KLighD	KIELER Lightweight Diagrams

KSbasE	KIELER Structure-based Editing
KWebS	KIELER Web Service
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MoC	Model of Computation
MVC	Model-View-Controller
MTM	Model-to-Model
NCSC	Normalized Core SCCharts
NOTEXP	Negate Expression
OAW	Open Architecture Ware
OREXP	Logical OR Expression
PAREXP	Parenthesized Expression
RCP	Rich Client Platform
REFEXP	Element Reference Expression
RELEXP	Logical Relation Expression
S	Synchronous
SC	Synchronous C
SC MoC	Sequentially Constructive Model of Computation
SCADE	Safety Critical Application Development Environment
SCG	Sequentially Constructive Graph
SCL	Sequentially Constructive Language
SCT	Yakindu Statechart Tools
SJ	Synchronous Java
SWT	Standard Widget Toolkit
UI	User Interface
WTO	Write-Things-Once

WTOTO WTO Transition Optimization

XMI XML Metadata Interchange

XML Extensible Markup Language

YSE Yakindu Statechart Editor

Chapter 1

Introduction

In general, designing and programming reactive and embedded systems is a difficult task. They impose stringent requirements and are often deployed in critical environments, e. g., in the automotive or aerospace industry. In the worst case the failure of such a critical system may result in the loss of lives.

Traditional programming languages are generally insufficient to model those critical systems due to their unpredictability. Therefore, synchronous languages which are strictly deterministic emerged.

1.1 Synchronous Languages

Synchronous languages are designed to ensure deterministic behaviour [BCE+03]. Especially the handling of concurrent control flows with interleaving dependencies is generally a challenging task. Different from the traditional programming paradigm of common languages such as Java and C. Concurrent threads in synchronous languages do not introduce *race conditions*, which are problematic with regard to ensuring determinism [Lee06]. Due to their predictability synchronous languages are particularly suited to model reactive and embedded systems. Especially, safety critical systems benefit from the synchronous approach. Figure 1.1 depicts the schematic of an embedded, reactive system encapsulated in a surrounding system.

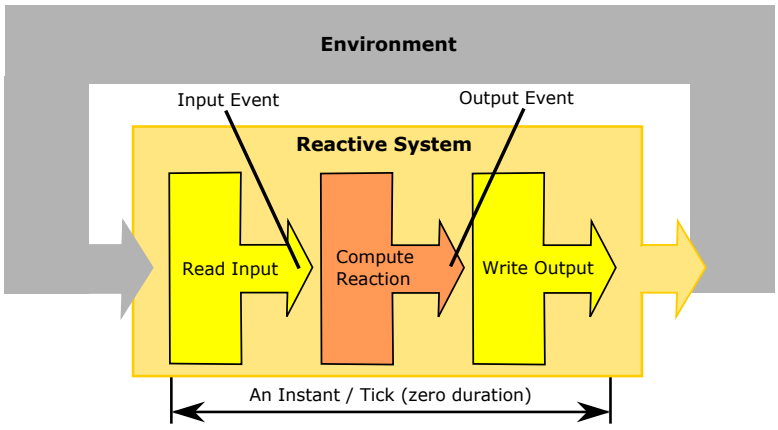


Figure 1.1. Embedded Reactive System [MvHH13]

In principle reactive systems interact constantly with their environment. They receive inputs from the surrounding environment, compute their outputs and feed the results back to the environment. In the classical synchronous Model of Computation (MoC) time is separated into discrete ticks and the computations the reactive system performs are considered to take no time. The discrete ticks, also known as *macro ticks*, consist of finite many single calculations, or *micro ticks*. This MoC is exemplified by languages such as Esterel, Lustre and SyncCharts [BCE+03] and is illustrated in Figure 1.2.

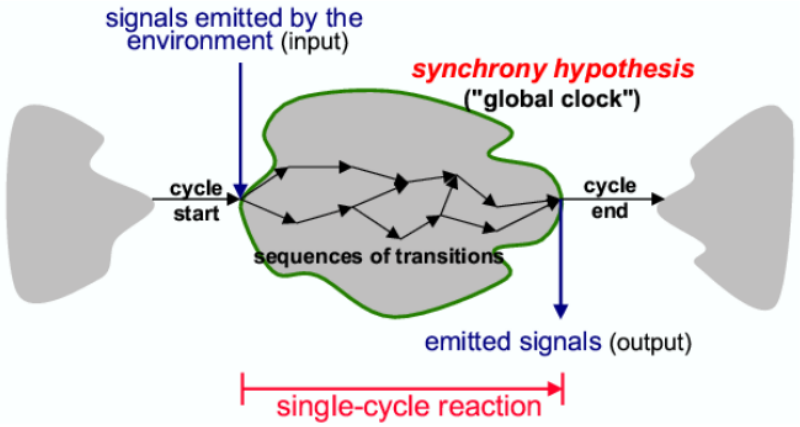


Figure 1.2. Synchrony Hypothesis (G. Luettgen, 2001)

The Synchrony Hypothesis

A system works in perfect synchrony if all reactions of the system are executed in zero time. Hence, outputs are generated at the same time when the inputs are read.

1.1.1 Sequential Constructiveness

Even though synchronous languages introduce the benefit of determinism to the repertoire of the programmer, they come with heavy restrictions on which programs are considered valid, or *constructive*. To ensure deterministic behaviour classical synchronous languages strictly forbid multiple different variable assignments in the same macro tick. Although it is possible to emit a signal more than once and combine the values with a predefined combination function, it is not permissible to emit the same signal more than once with different distinct values.

The Sequentially Constructive Model of Computation (SC MoC) introduced in "Sequentially Constructive Concurrency - A conservative extension to the synchronious model of computation" [vHMA+13b] by von Hanxleden et al. in 2012 extends the classical synchronous MoC by lifting some of these restrictions. The SC MoC allows variables to be read and written multiple times in the same tick instance as long as the program

stays sequentially schedulable. As a result common familiar programming paradigms are available again without sacrificing determinism [vHMA+13b].

1.1.2 SyncCharts

In 1996 Charles André introduced the Statecharts formalism, proposed by David Harel in 1987 [Har87], to the synchronous world [And96]. SyncCharts can be seen as graphical representation of the synchronous textual language Esterel which is used to describe reactive systems [BC84].

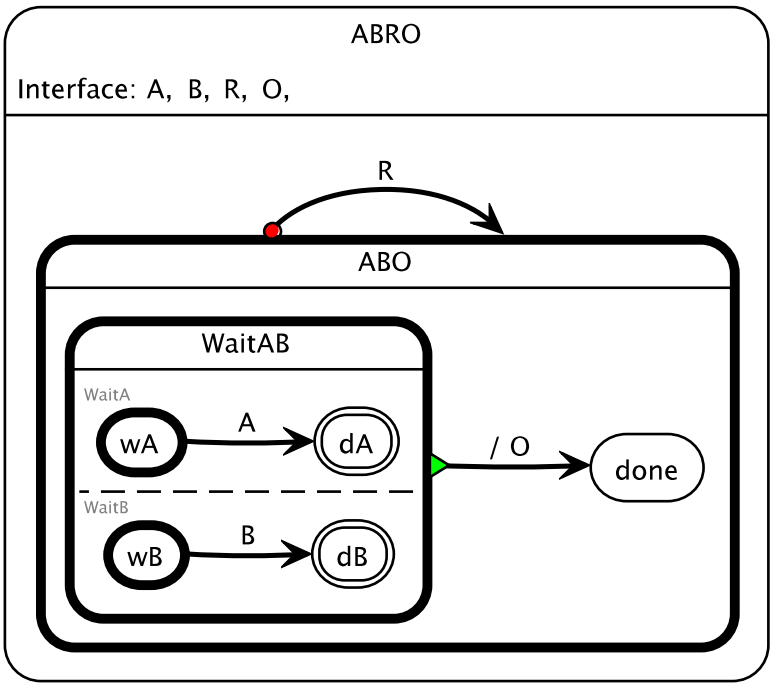


Figure 1.3. ABRO, the "Hello World" of SyncCharts

SyncCharts Elements

Figure 1.3 shows the ABRO program, the common "Hello World' of Esterel, and depicts the most typical SyncCharts elements.

Signals: Signals are the main communication mechanism in SyncCharts. A signal can either be *present* or *absent*, but not both at the same instance. As shown in Figure 1.3 signals are defined in the interface section of a composite state. The most top level state, the *main state*, holds the interface to the environment. Here, signals might be defined as *input* or *output* signals. Input signals are information coming from the

environment whereas output signals represent the final computations of the system which are given back to the environment. Signals may be defined with a certain type (boolean, integer, etc.) or as *pure* in which case they do not hold any additional information.

States: SyncCharts are built out of states. If a state is illustrated with a thick border, it is marked as *initial* state. An initial state becomes active if his parent region is entered. *Final* states are depicted with a double border. The containing region will terminate, if such a state is reached. A state itself can be a composite state, also called *macro state*, containing one or more regions which themselves can contain a group of states again.

Regions: A region is a collection of states. Since several regions can be active at the same time, they introduce parallelism to SyncCharts. They must include a state marked as initial.

Transitions: Transitions are used to travel from one state to another. They can be either *weak* in which case the actions of containing regions are executed before the transition takes place or may be *strong*. Strong transitions, denoted with a red dot, pre-empt macro states and proceed at once. As a third transition type a *normal termination* serves as join for concurrent regions. The normal termination is executed if all preceding regions are in a final state.

The Signal Coherence Law

Within one tick a signal can either be present or absent but not both at the same time. A signal S is absent by default and only present in a tick, if and only if S is emitted in any active transition in this tick.

1.1.3 SCCharts

The synchronous language SyncCharts exemplifies the classical synchronous MoC. Therefore, it can be understood as a graphical representation of Esterel. Similar to the extension of the classical synchronous MoC by the SC MoC, the statechart dialect SCCharts extends SyncCharts and employs the SC MoC. An SCChart is considered constructive if it is syntactically correct and sequentially schedulable. Since the SC MoC is a conservative extension, the class of SCCharts includes the class of SyncCharts.

SCCharts come in two variations. *Core* SCCharts are composed of a minimal set of elements which are necessary to express all features of the SC MoC such as simple states, transitions and hierarchy. However, *Extended* SCCharts add common features and syntactical sugar to their core relatives to simplify the creation of complex models. Every Extended SCChart can be translated to a semantically equivalent Core SCChart [vHMA+13b].

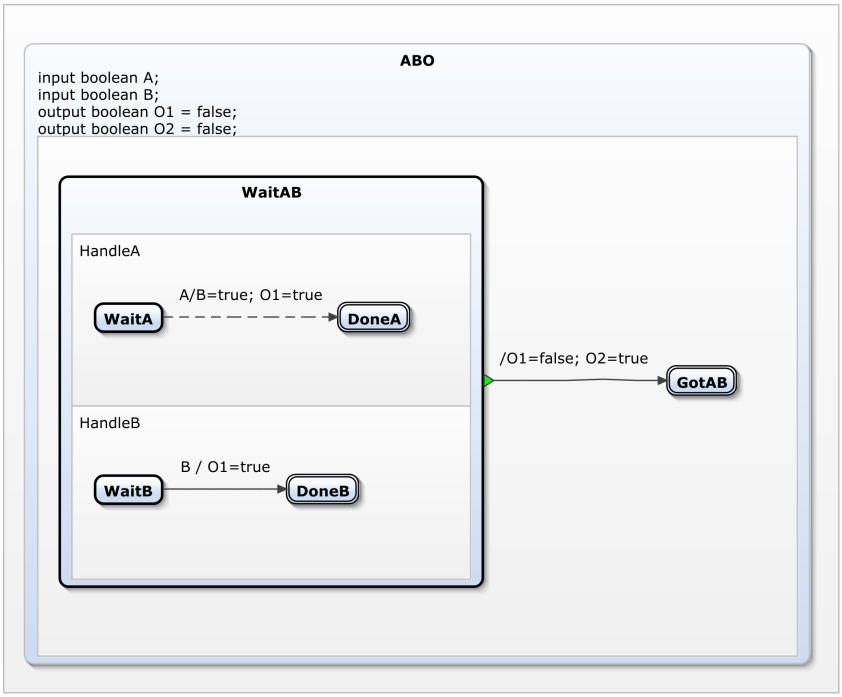


Figure 1.4. ABO, the "Hello World" of SCCharts

Figure 1.4 shows the ABO example, the "hello world" of the sequentially synchronous world. It is a Core SCCharts and demonstrates the common hierarchical layout with states, transitions including triggers and effects. The dashed edge resembles an *immediate transition* and the green triangle symbolizes a *normal termination*, a join of concurrent threads. Detailed information on each SCChart element is provided in Section 4.1.1.

1.2 Model-driven Development with KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] is a research project developed at the Real-Time and Embedded Systems Group at the Christian-Albrechts-University in Kiel, Germany. The main focus of the project is to develop new concepts and methods for designing and editing different types of diagrams. Essentially, the KIELER framework is a set of open source Eclipse plug-ins which integrate with common Eclipse modelling projects, such as the Eclipse Modeling Framework (EMF). It consists of over 170 plug-ins that are made available in 19 features and can safely be considered as a large Eclipse-based project [Gre12].

As illustrated in Figure 1.5 KIELER is separated in distinct areas. The *semantics* area focuses on simulation of and code generation for graphical and textual modelling

[1] http://www.informatik.uni-kiel.de/kieler

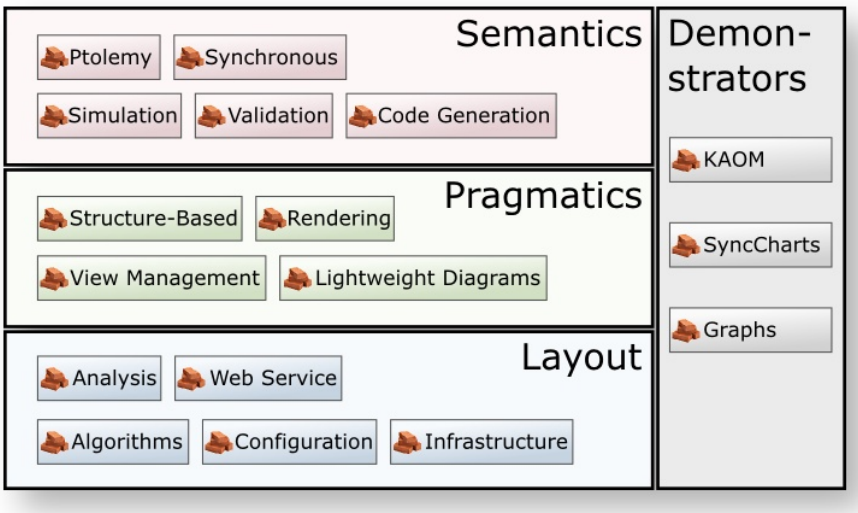


Figure 1.5. KIELER Project Structure (KIELER Documentation)

languages with emphasis on synchronous languages. *Pragmatics* provides new and simple ways to edit and visualize graphical representations. The *layout* section provides features for the automatic layout mechanisms. As a fourth segment the *Demonstrators* part gathers editors used to explore and demonstrate the technologies developed in the three main areas mentioned before.

The approach presented in this thesis is embedded in the KIELER project and thus, benefits from all existing tools in the framework. Although it is primarily located in the semantics area since the main aspect of this thesis deals with code generation, the pragmatics and layout areas also grant advantages with respect to information visualization.

1.3 Problem Statement

Although synchronous languages simplify the modelling of concurrent behaviour, writing such a program manually becomes increasingly difficult as the complexity of a given issue rises. Dealing with concurrency pitfalls can easily lead to frustration and unpleasant bug searching sessions while the real modelling problem remains untouched. Graphical editors aim to ease the effort necessary to create such programs even more, but the created diagrams have to be translated to code. Automatically generated code often is unoptimized or hard to read which makes analysing and debugging difficult. Even though synchronous code can be executed and validated in simulators specifically designed for a synchronous environment, real-time and embedded systems often comprise components which only support traditional programming and languages such as C.

1.4 Outline of this Document

The solution presented in this thesis will describe in detail how to automatically generate code originating from SCCharts. The translation is made in several key steps.

Prior to starting with the explanation of the concept, indicated in Section 1.3, Chapter 2 discusses general code generation concepts with emphasis on the synchronous world. Additionally, it explains the code generation in SyncCharts. SyncCharts are used as main comparison languages to the approach presented in this thesis. The chapter closes with a second example, SCADE which is also a graphical modelling and code generation tool used in the industry.

Chapter 3 introduces the technologies used throughout the implementation of the approach. As this approach is embedded in the KIELER project and KIELER itself is an Eclipse project, it gives an overview of the Eclipse project with emphasis on the modelling frameworks such as EMF, Graphical Modeling Framework (GMF) and Graphical Editing Framework (GEF). Additionally, the chapter introduces the modelling languages Xtend[2] and Xtext[3]. Furthermore, it describes the KIELER tools used in the approach to layout and visualize the Sequentially Constructive Language (SCL) such as KIELER Layouters (KLay), KIELER Lightweight Diagrams (KLighD) and KIELER Execution Manager (KIEM). The chapter closes with the illustration of the Yakindu framework, a modular framework used for editing statecharts recently implemented in KIELER.

Chapter 4 presents the chain of key steps of the code generation approach as indicated in Section 1.3. The chapter is separated into two parts. In the first part all mandatory language concepts are presented. At the beginning SCCharts and the general concept of sequentially constructiveness are explained. Then, the SCL and its graphical representation the Sequentially Constructive Graph (SCG) are illustrated in detail. Additionally, the restrictions of Sequential SCL are exemplified. Subsequent to the illustration of the language concepts the second part of the chapter exemplifies the transformations necessary to proceed from one key step to another. It describes the transformation from SCCharts to SCL and the synthesis of the corresponding SCG. The chapter presents how the SCG can be used to evaluate the generated SCL programs and how to identify potential problems during the code generation. Eventually the sequential tick function can be engineered. To generate such a tick function and resolve potentially concurrent conflicts, the control flow of the SCL program has to be analysed. The second half of the transformation part illustrates how this program examination is done. As this tick function is also an SCL program, all previously acquired methods can be be used to analyse this generated program. The chapter closes with the transformation to the Synchronous (S) program which is necessary for the simulations and evaluations in KIELER and ideas to go further from here.

Chapter 5 describes the key implementations of the concepts presented in Chapter 4. The chapter begins with the modelling of the SCL metamodel. It continues with the

[2] http://www.eclipse.org/xtend/
[3] http://www.eclipse.org/xtext/

implementations of SCL extensions which enhance the developed model. Furthermore it explains the realization of the SCG synthesis. The chapter concludes with the implementation of the Model-to-Model (MTM) transformations used to translate the models according to the compile chain of the approach.

Chapter 6 comprises the evaluation of the concept presented in Chapter 4. Therefore, it describes the simulation set-up and compares the results of the approach with SyncCharts, introduced in Section 1.1.2. The evaluation mainly focuses on the execution times and summarizes results of different test runs with distinguished test settings.

The last chapter, Chapter 7, recapitulates the approach presented in this thesis and gives an outlook on potential future work.

Chapter 2

Related Work

Statecharts are particularly well suited for the modelling of reactive systems. However, generating efficient code automatically is a difficult task. Firstly, this chapter explains general code generation concepts and their usefulness with respect to the synchronous world.

Secondly, an already established code generation approach "SyncCharts to Synchronous C (SC)" is explained. Since SyncCharts are included in the KIELER framework, they are used to compare the presented approach. Besides the comparison of language concepts, it is also used in the experimental results to evaluate the outcome of the code generation implementation presented in this thesis.

Furthermore, the popular Safety Critical Application Development Environment (SCADE) is presented as another example for a Model-Driven Software Development (MDSD) environment. It is particularly designed for safety-critical applications and well-established in the industry.

2.1 General Code Generation Approaches

In principle, three code generation concepts for statecharts exist.

Ali and Tanaka [AT00] describe the transformation from states into classes at the example of Java. The approach can be categorized as *pattern translation* since each type of state is transformed into code according to an defined pattern.

The second concept, presented by Wasowski [Was03], translates statecharts elements into hierarchical trees. These trees are used to calculate the control flow.

The arguably most simple way to translate a statechart is the translation into a Finite State Machine (FSM). Wasowski describes in "Flattening statecharts without explosions" [Was04] how hierarchical regions of stateschart are eliminated efficiently. The disadvantage of this approach lies in the exponential growth of the FSM.

However, in "Synthese von SC-Code aus SyncCharts" [Ame10] Amende already pointed out that general code generation approaches are not sufficient for SyncCharts due to the nature of their semantic differences. As stated in Section 1.1.3, since SyncCharts are included in the class of SCCharts, this is also true for SCCharts.

The approach presented in this thesis orientates on the approach described by Amende which can be categorized as *simulation based appoach*. However, as the transformation chain of this approach comprises several distinct key steps, each step may also incorporate features of the other concepts such as pattern matching.

2.2 SyncCharts Code Generation

As mentioned in Section 2.1 Amende presented a code generation approach for SyncCharts, introduced in Section 1.1.2, in 2010. The compiler creates a dependency tree which contains all dependencies between signals, control flow and hierarchies. If this tree is acyclic, it can be sorted and determines the execution sequence of the generated code. Subsequently, calculated thread priorities preserve the deterministic nature of the SyncChart. The close relationship between SyncCharts and the target language SC facilitates the translation. [Ame10]

This approach is similar but not identical to the approach presented in this thesis since determinism is preserved through priorities. The code generation explained in this thesis provides determinism through the ordering of statement amalgamations, so called Basic Blocks (BBs). Although these blocks also require acyclic dependencies, the concepts differ. The concept behind this approach will be elucidated in Chapter 4.

However, as explained in Section 1.1.3, SCCharts make a significantly larger of class of statecharts acceptable without compromising determinism. Furthermore, SCCharts use shared variables instead of signals but these can be fully emulated with variables under sequentially constructive scheduling as will be shown in Section 4.1.2.

2.3 SCADE Code Generation

SCADE is a MDSD environment developed by Esterel Technologies[1] and especially designed for safety-critical application development. It is well-established in the industry. The Integrated Development Environment (IDE) includes an editor for model creation, a model verifier and a code generator.

SCADE uses a variant of SyncCharts elements for code generation to augment diagrams with reactive behaviour by extending boolean clocks towards clocks that express state [CPP05] [CHP06].

Analogously to SyncCharts, the variant of SyncCharts in SCADE is restricted to constructiveness in Berry's sense [Ber99]. Therefore, the approach presented in this thesis accepts a larger class of statecharts.

[1] http://www.esterel-technologies.com

Chapter 3

Used Technologies

Prior to explaining the main ideas of the approach and their implementation details, it is necessary to present some key technologies used throughout the next chapters. Even though some of this technologies have already been mentioned in the previous chapters this chapter is going to introduce them in detail.

3.1 The Eclipse Project

Since the approach presented in this thesis is contained in Eclipse plug-ins and is designed to be executed in the KIELER framework, a closer look at Eclipse with emphasis on the modelling tools is necessary.

The Eclipse Project[1] is an open source software application introduced by IBM in 2001. It is developed in Java and commonly known as *the* Java IDE. Even though it may be one of the most common IDE for developing in Java, Eclipse is designed as open component-based software system and can be used in many different ways, e.g. software development environments in other languages such as C or as diagram modelling tool.

The generated Eclipse Rich Client Platform (RCP) consists of basic core components and plug-ins to extend the functionality of the RCP. The core components manage main areas such as the workbench, the help and update centers, file system and the runtime kernel whereas the implementation of specific tasks is contained in the plug-ins.

When working with Eclipse RCPs a common *look-and-feel* is carried throughout different projects. Many structures such as toolbars, editor features and resource management can be found in these applications. This amalgamation of tools is called the Eclipse *workbench*, depicted in Figure 3.1. It provides a collection of elements which cover the basic needs of programming RCP applications. This includes *editors*, *Views*, *Perspectives* and *Wizards*.

An editor in Eclipse is an editing component which can be used to browse and change content of resources. The purpose of an editor can be generic and associated with multiple resources such as text editors or it can be specific and designed for one Domain Specific Language (DSL) only. Furthermore, views are used in many ways. The information shown depends on its purpose, e. g., status or debugging information, property selection and outlines.

[1] http://www.eclipse.org

A perspective is a set of visible editors and views and their location. They have generally a specific purpose. Common perspectives are e. g., Java developing and debugging. Only one perspective can be active at any time.

Wizards are dialogs an user can use to enter task specific information. The project creation wizard for example enables the user to specify workspace location and name of a new project and then creates this project automatically.

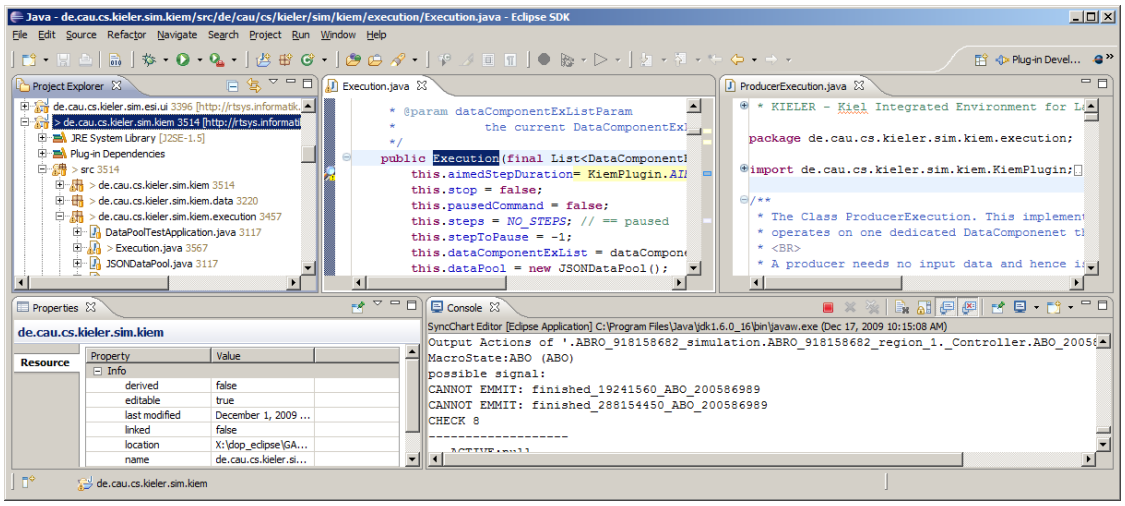


Figure 3.1. Eclipse Workbench with Editors and Views [Mot09]

3.1.1 Graphical Editing Framework

The Graphical Editing Framework (GEF)[2] provides technology to create rich graphical editors and views for the Eclipse project. It is composed of three components: Draw2d, GEF MVC and Zest.

Draw2d is a layout and rendering tool-kit for displaying graphics on a Standard Widget Toolkit (SWT). Graphical components in Draw2d are called *figures* and represented by simple Java objects. These objects can be composed via a parent-child relationship.

GEF MVC is an interactive Model-View-Controller (MVC) for the GEF.

Zest is a visualization tool-kit based on Draw2D, which enables implementation of graphical views for the Eclipse project.

3.1.2 Eclipse Modeling Framework

Since this model-driven approach utilizes models to handle and visualize all mandatory data, a model framework is required. The Eclipse Modeling Framework (EMF) is *the*

[2] http://www.eclipse.org/gef

commonly accepted model framework for Eclipse. EMF consists of three fundamental pieces: the *core EMF framework*, *EMF.Edit* and *EMF.Codegen*.

The models used by the EMF must follow an abstract syntax. These abstract model descriptions are called *metamodels*. The core EMF framework provides tools to create metamodels and runtime support for these models, including change notification, XML Metadata Interchange (XMI) serialization and an Application Programming Interface (API) for manipulating EMF models. Models created with this framework are called *Ecore models*. Since the core EMF framework is also an Ecore model, all provided tools are applicable to the EMF itself.

EMF.Edit is a framework that includes generic classes for building editors for EMF models and the EMF.Codegen facility can be used to create whole functional editors for EMF models including a complete Graphical User Interface (GUI).

Metamodels

To define the abstract structure of a model, the EMF uses descriptions which are again models. These EMF models are represented by a language called *Ecore*. Therefore, Ecore is a EMF model for other models, or metamodel. A model which follows the rules of a specific metamodel can be considered a *metamodel instance* of the corresponding metamodel.

Metamodels consists of *classes*. A class is identified by its name and may contain *attributes* and *references*. An attribute must have a name and a *data type* whereas a reference is an association between classes. The dependencies of the Ecore elements are illustrated in Figure 3.2.

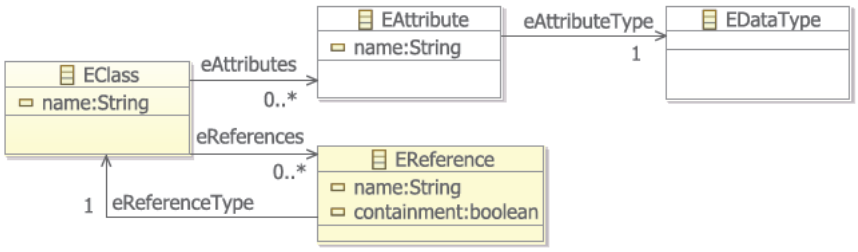


Figure 3.2. Simplified Ecore Metamodel Subset [Mot09]

A metamodel can be created with the EMF tree editor or with the integrated graphical editor. It is also possible to import a metamodel from an Extensible Markup Language (XML) schema definition or to derive it from a Java implementation. EMF classes including their attributes and references can be created in the tree editor. The properties of the members can be changed in the associated *property view*. Figure 3.3 illustrates the editing of a EMF model. A class transition is selected and its super types, indicated by the right

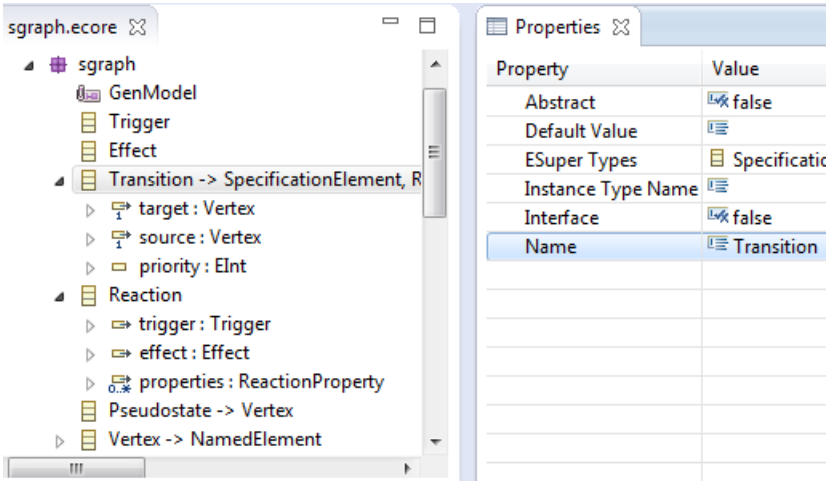


Figure 3.3. EMF Tree Editor [Har13]

arrow, references and attributes are shown. The graphical editor enables the creation of a metamodel in an intuitive way. Dependencies and relationships between all classes are visible and editable. An example of graphical EMF editing is depicted in Figure 3.4.

Once the metamodel is completed, the EMF generator can be invoked to generate all mandatory Java classes for this model. The created source code already includes classes for parsing and serializing model data. In general, EMF models are saved as XMI resource. The generated classes contain fields, setter and getter for all defined attributes and references.

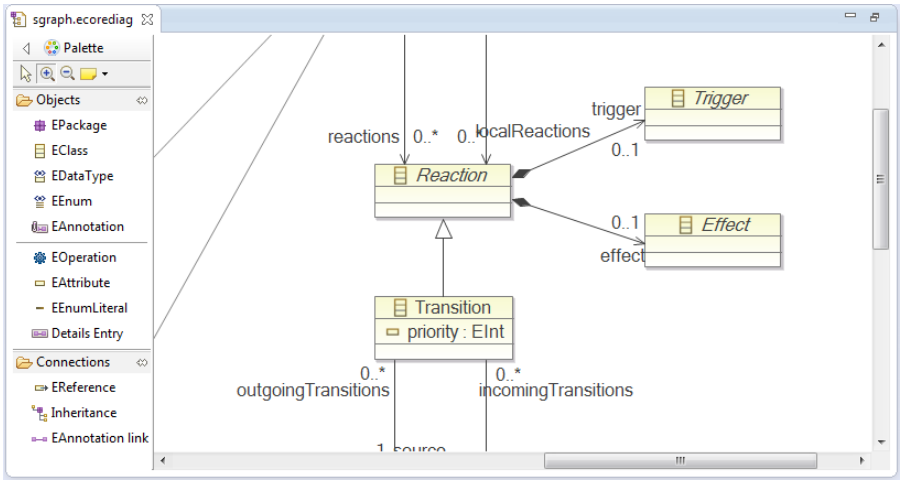


Figure 3.4. EMF Graphical Editor [Har13]

3.1.3 Graphical Modeling Framework

The Graphical Modeling Framework (GMF)[3] closes the gap between EMF and GEF. It uses the GEF to display graphic elements called *figures* which resemble model elements created with the EMF. Figures are basically nodes and edges or a composition of both of them. A GMF based editor is equipped with many tools for creation of the desired diagram. Mechanisms for editing figures, positioning, loading and saving are available out of the box. A diagram is saved in form of a *notation model* to preserve the structure of the model as well as the positions and sizes of the figures.

An editor can be developed manually by utilizing the GMF runtime which comes with many features such as a set of reusable components for graphical editors, a command infrastructure and an extensible framework which allows graphical editors to be extendible. However, the GMF tooling provides an alternative way of creating a graphical GMF based editor automatically. By defining graphical and model mapping definitions, a fully functional graphical editor can be derived based on the GMF runtime.

3.1.4 Xtext

A powerful way to create a DSL is provided by Xtext[4], a framework for development of programming languages and DSLs. It was published by the itemis AG[5] in 2006 as part of the former code generation framework Open Architecture Ware (OAW) and is based on the EMF.

The programmer defines a grammar of his DSL in Xtext's textual language as depicted in Listing 3.1. Subsequently, the framework generates the complete language infrastructure including a corresponding metamodel, a parser, a serializer and an editor. It comes with good defaults for all these aspects and at the same time every single aspect can be tailored to the needs of the modeller. A grammar can be derived from other grammars and usually consists of *rules*. A rule can either be an extension to former rules or a *terminal rule*. The rule definition symbols, which are semantically similar to regular expressions, determine the cardinality of data fields.

No character: exactly one

Question mark (?): none or one

Plus sign (+): at least one

Asterisk (*): arbitrary many

Furthermore, the data type assign operator defines the kind of the created field.

Simple equal (=): the field has a single data type

Add operator (+=): the field is a list of a specific type

Boolean assignment (?=): the field is a boolean flag

[3] http://www.eclipse.org/modeling/gmp
[4] http://www.eclipse.org/Xtext/
[5] http://www.itemis.de

```
grammar de.cau.cs.kieler.scl.SCL
    with de.cau.cs.kieler.yakindu.sccharts.model.stext.Synctext

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate scl "http://www.cau.de/cs/kieler/scl/scl"

Program:
    'module' name = ID
    (definitions+=VariableDefinition)*
    '{'
    (
        ((statements += InstructionStatement';') | statements += EmptyStatement)*
        (statements +=  InstructionStatement statements += EmptyStatement*)?
    )
    '}'
;
```

Listing 3.1. Xtext Grammar Example

3.1.5 Xtend

The MTM transformation language Xtend[6], which also was introduced by the itemis AG as part of the OAW, integrates with the EMF. It is a statically-typed programming language which translates to comprehensible Java source code. Although many aspects of Xtend are Java-like, a lot of features and syntactical sugar with emphasis on advanced queries on model elements have been added to the language. The Xtend Listing 3.2 translates to Listing 3.3.

```
class HelloWorld {
    def static void main(String[] args)
        println("Hello World")
    }
}
```

Listing 3.2. Xtend Example (Xtend Documentation)

```
// Generated Java Source Code
import org.eclipse.xtext.xbase.lib.InputOutput;

public class HelloWorld {
    public static void main(final String[] args) {
        InputOutput.<String>println("Hello World");
    }
}
```

Listing 3.3. Generated Java Example Code (Xtend Documentation)

[6] http://www.eclipse.org/xtend/

According to the Xtend documentation model, transformations added to existing types which do not modify the model are called *extensions*. Xtend comes with many generic extensions such as *map*, *filter* and *sort*. All MTM transformations and model extensions in this thesis are implemented with Xtend.

3.2 Kiel Integrated Environment for Layout Eclipse Rich Client

As mentioned in Section 1.2 the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[7] is a academic research project about enhancing graphical, model-based design developed at the Real-Time and Embedded Systems Group at the Christian-Albrechts-University in Kiel, Germany. It is developed as open source software licensed under the Eclipse Public License (EPL) and attempts a tight integration with the Eclipse modelling projects such as EMF, GMF, Xtext, etc. As depicted in Figure 1.5 the project is separated in four areas.

Semantics

Since the approach described in this thesis addresses the field of code generation this work is mainly situated in the semantics area. KIELER semantics also provide an infrastructure to define execution for metamodels and different approaches to simulate models using simulators based on C, Java or Ptolemy[8]. These simulations are integrated via the KIEM and can easily be extended.

Pragmatics

The KIELER pragmatics group aims to simplify the daily work of modellers in the context of Model-Driven Engineering (MDE). As depicted in Figure 1.5 this includes projects such as KIELER Structure-based Editing (KSbasE), KIELER View Management (KIVi) and KIELER Lightweight Diagrams (KLighD).

KSbasE is a structure-based editing extension that provides operations for modifying models. Elements can be added, deleted or connected to other components with a few mouse clicks. [Mat10] KIVi focuses on dynamic visualizations in diagrams. In principle it receives notifications fired by certain *triggers* and reacts with corresponding *effects*, e. g., performing an automatic layout. Triggers can be bound to effects over a *combination* mechanism enabling the programmer to customize the behaviour of specialized views for specific models. [Mül10]

[7] http://www.informatik.uni-kiel.de/kieler
[8] http://ptolemy.eecs.berkeley.edu/

Layout

The Layout section enables one of the profound ideas of KIELER, freeing the user from time-consuming model preparation tasks such as moving nodes and re-routing edges. The designer is instantly capable to do the change and must not focus on layout questions.

The core component of the KIELER layout service is KIELER Infrastructure for Meta Layout (KIML). It provides the connection between graphical editors and layout algorithms. Besides commonly known open source layout libraries such as OGDF and Graphviz, KIELER provides its own custom layout algorithms in KLay.

Additionally many layout services are accessible over the KIELER Web Service (KWebS) component.

Demonstrators

Demonstrators is a general term for all editors used in KIELER to demonstrate and test ideas and results of the three main sections. The approach presented in this thesis utilizes two of these demonstrators, the KIELER SyncCharts editor ThinKCharts and the Yakindu Statechart Editor (YSE) editor.

3.2.1 KLay Layered

One of KIELER's most advanced custom layout algorithm is KLay Layered. It was introduced by Spönemann in 2009 in "On the automatic layout of data flow diagrams" [Spö09] and further optimized by Schulze in 2011 in "Optimizing Automatic Layout for Data Flow Diagrams" [Sch11]. It is primarily used for layouting data flow diagrams and supports hierarchical *node placement*, *ports* and comes with a set of customizable properties.

3.2.2 KLighD

KIELER Lightweight Diagrams (KLighD) is part of the KIELER pragmatics area and offers a transient representation of models without incorporation of complex editing facilities. In this *transient view* approach the model is constructed interactively as the modeller works with an arbitrary editor, e. g., based on a textual DSL. In addition to this simplified way to visualize an arbitrary model graphically changes to the model are incorporated and displayed instantaneously. [SSvH12a]

KLighD uses the piccolo toolkit[9] for visualization and comes with powerful generic extensions for synthesizing models into corresponding graphical representations. Once a specific synthesis maps the model elements to figures the view is almost instantly available.

[9] http://code.google.com/p/piccolo2d/

3.2.3 KIEM

Responsible for simulating models in KIELER is the KIELER Execution Manager (KIEM). It provides the main simulation infrastructure and specifies the interface for other plug-ins. Essentially KIEM is a communication bus between independent and concurrent components. These so called *data components* observe information on the communication bus or produce own data, which can then be used again by other components or simply displayed in a view [Mot09]. The KIEM bus schematic is illustrated in Figure 3.5.

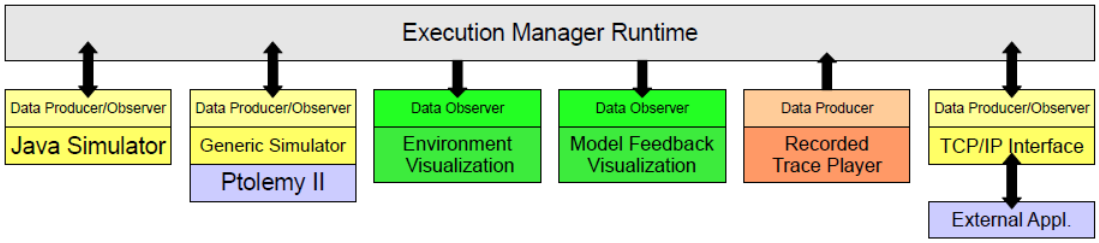


Figure 3.5. Schematic Overview of the KIEM Interface [MFvH10]

3.2.4 Synchronous

Essentially, Synchronous (S) is the common basis of Synchronous C (SC) [TAvH11] and Synchronous Java (SJ) [MvHH13]. Both combine the synchronous paradigm of SyncCharts, including determinism and pre-emption, with the commonly supported languages C and Java. S is designed as intermediate language and origin for further transformations and simulations. Models translated to S can be simulated in KIEM instantaneously or compiled to executable SC or SJ source code.

Since S is mainly developed to provide a common basis for statechart dialect, it comprises an abstract syntax similar to the syntax of statecharts in textual form. Analogously to statecharts, the behaviour of the chart is included in *states*. These are permitted to traverse to other state or change the priority of a concurrent context to allow rescheduling. They also utilize signals equally to the signals in SyncCharts described in Section 1.1.2 as main communication mechanism and contain a signal interface for their environment.

3.3 Yakindu Statechart Editor

The Yakindu Statechart Editor (YSE) is part of the Yakindu Statechart Tools (SCT), a collection of Eclipse-based tools designed for the development of reactive and event-driven systems. SCT is part of the Yakindu Open Source developed by the itemis AG[10]. The editor is based on the GMF and therefore simple to adapt to custom requirements. It provides

[10] http://www.itemis.de/

two derivable metamodels for the definition of user-defined abstract syntaxes. *SGraph* describes the syntax of the graphical elements whereas *SText* specifies the expression language used in the statecharts. [Har13]

The root of an SGraph is typically a *statechart* and comprises *regions*. A region contains *vertices*. One specialized type of a vertex is a *state*. States may be marked as *initial* or *final* state. They can be a *composite* element and are therefore able to enclose new regions. Generally, *transitions* connect states.

The implementation of the SCL presented in this approach utilizes the expression language defined by *SText*. The two most important model objects utilized by the SCL are the *Element Reference Expression (REFEXP)* and the *Assignment expression (ASSEXP)*. The first one binds an expression object to a model element definition. The latter describes an assignment of an expression. Furthermore, the expression of an assignemnt can be contained arbitrarily in *Parenthesized Expression (PAREXP)* and common operators. For instance, a PAREXP may contain a collection of *Logical AND Expression (ANDEXP)* which connects the references with a logical "and". Also noteworthy in this context is the *NegateExpression* which negates the referenced expression.

3.3.1 KIELER SyncCharts Editor based on Yakindu

For further SyncChart projects and thus also as starting point for this thesis the YSE was adapted to support SyncCharts and SCCharts by Haribi in "A SyncCharts Editor based on Yakindu SCT" [Har13]. Since then several enhancement to the editor have been implemented. The derived new KIELER Yakindu implementation incorporates the following features.

transition types: There are three different types of transitions. *Weak* and *strong* transitions and *normal terminations*. A strong transition preempts the actual and all embedded states. On the contrary a weak transition does not pre-empt the actual state, but proceeds to its target after the execution of the current tick. Normal terminations resemble a *join* of subsidiary threads.

immediate transitions: Usually, in the context of the sequential MoC a transition from one state to another consumes time. In this case the execution of the model is paused until the next macro tick starts. To continue the execution at once a transition can be marked as *immediate*. These transitions are illustrated as dashed edge in the graphical model. Immediate transitions may be denoted with a hash tag (#) in different editors.

expression improvements: Transition *trigger* may have a priority notation at the beginning of the trigger. The evaluation of priority changes is done by KSbasE. Additionally, unlike to the original Yakindu expressions guards do not need brackets as delimiter. Lastly, the KIELER expression extension supports arbitrary encapsulated *pre* and *val* function calls.

Chapter 4

Sequentially Constructive Code Generation

This chapter is separated in two main sections. Section 4.1 introduces mandatory language concepts used throughout the approach including sequential constructiveness in general, whereas Section 4.2 describes the ideas behind the particular transformations. The code generation concept presented in this thesis proceeds along several key steps originating from SCCharts.

In the first step Extended SCCharts are translated into their core equivalent to build the basis for the subsequent SCL transformation. Once the SCL program is generated the corresponding SCG can be synthesized. After performing a dependency and a basic block analysis to gather information about the schedulability of the program it is possible to create a generic tick function. It is a sequential SCL program stripped of concurrent threads and registers and is meant to be an initial point for further software or hardware

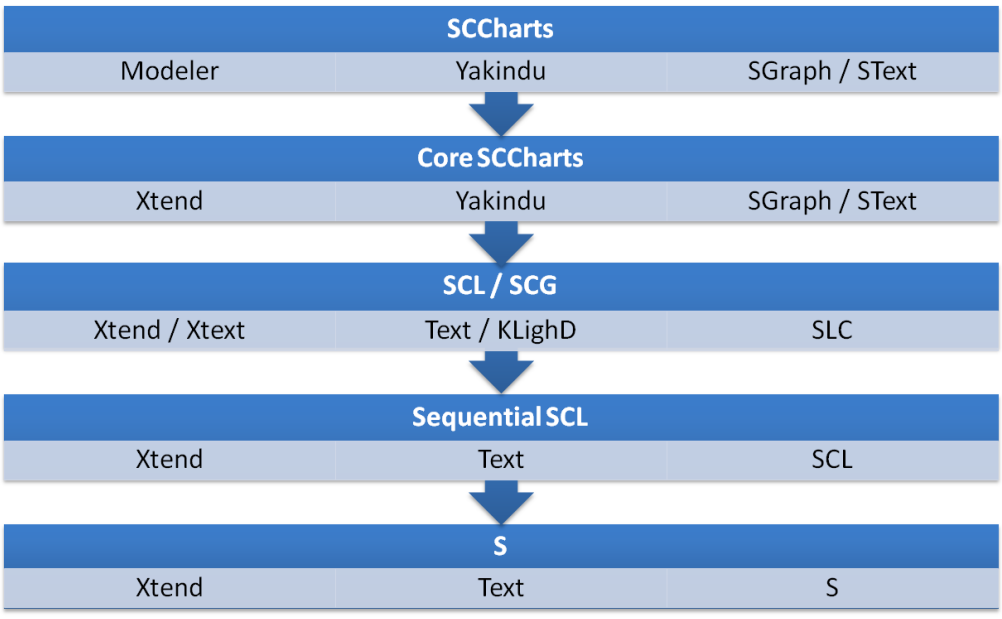


Figure 4.1. Transformation chain including origin and underlying metamodel

synthesis. Lastly, the SCL tick function is translated to S to perform simulations in KIELER and to compare the results with other approaches. Figure 4.1 illustrates the key steps of the approach. The origin of the transformation method is listed on the left hand side. The middle section depicts the visualization mechanism and the underlying metamodel is shown on the right side.

4.1 Language Concepts Introduction

To understand the different transformation steps a closer look at the languages used in the approach is necessary. The following sections will describe SCCharts, their two variants Extended and Core SCCharts, SCL and its graphical representation SCG in detail.

4.1.1 SCCharts

SCCharts extend SyncCharts, elucidated in Section 1.1.2, and employ the SC MoC. Instead of using *signals*, introduced in Section 1.1.2, as main communication mechanism SCCharts possess *typed variables*, which can be read and written multiple times in one macro tick as long as the program stays schedulable. Consequently the heavy restriction of the signal coherence law, defined in Section 1.1.2, is partly lifted. Hence, SCCharts combine the determinism of the synchronous MoC with sequential programming techniques known from common languages such as C and Java.

The common programming constructs shown in Listing 4.1 and graphically illustrated in Figure 4.2. It is forbidden according to the classical synchronous MoC, since I is set to true, when it is false, which contradicts the signal coherence law. However, considering the assignment being perfectly schedulable, the example is constructive in SCCharts.

```
1 if not I then I = true end
```

Listing 4.1. Basic valid sequential Assignment

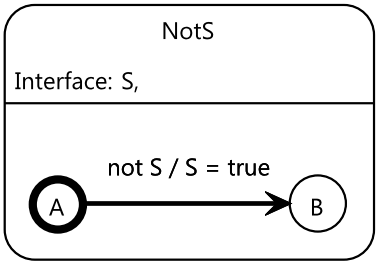


Figure 4.2. Basic valid SCCharts Example

As mentioned in the introduction of this chapter SCCharts come in two variations, *Core* SCHarts and *Extended* SCCharts. Core SCCharts provide the mandatory elements to form constructive statecharts with regards to the SC MoC, whereas Extended SCCharts comprise more powerful expressions to model compact diagrams, but are semantically equivalent to their core relatives. Both are exemplified in detail in the next two sections. The absence of signals in SCCharts is no disadvantage since they can be fully emulated with boolean variables and are in fact reintroduced in Extended SCCharts.

As final general SCCharts example Figure 4.3 depicts ABO, the "Hello World" of SCCharts. Though there is a strong resemblance to SyncCharts, the output variable O1 might be written several times in one macro tick. If A is present, B and O1 will be emitted and set to true. Consecutively O1 will be set to true again in HandleB because B is true. Both regions are now in their final state and will trigger the Normal Termination, which executes immediately and sets O1 to false in the same tick instance. Just like the NotS example this is a constructive SCChart.

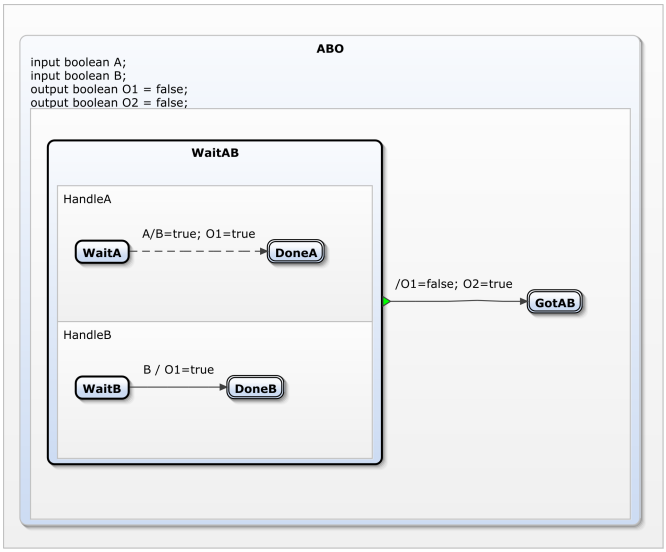


Figure 4.3. ABO, the "Hello World" of SCCharts, in detail

Core SCCharts

The fundamental element set of SCCharts is called *Core* SCCharts. Figure 4.4 illustrates all SCCharts elements. It is divided in two regions. The upper region presents the core elements, whereas Extended SCCharts figures are shown in the bottom region. Core SCCharts consist of the following elements only:

States: States in SCCharts behave analogously to the states in SyncCharts. They may be marked as *initial* or *final*. A state holding one or more regions is called a *macro state*. If a state is not a macro state, it is called *simple state*.

Regions: Regions are identical to their counterpart in the SyncCharts language and provide the modeler with parallelism.

Variables: As discussed in Section 4.1.1 Core SCCharts do not make use of signals. The main instruments for communication are typed variables. Similar to the signal declaration, variables may be declared as *input* or *output variable* and may have an *initial value*. Additionally they can be defined as *local*, which makes them valid in their macro state context only. A local variable is also allowed to be *static*, meaning its value persists even, when its context is re-entered. In contradiction to signals they can be read and written multiple times in any macro tick, provided they follow the rules imposed by the SC MoC.

Transitions: Practically different transition types must not be distinguished in Core SCCharts. The type of any transition depends on its source state. If it is a macro state, the transitions must be a Normal Termination. They are triggered if all concurrent regions in the preceding state are in any final state. Since simple states do not have containing regions an outgoing transition will be a weak transition allowing any action in this tick to be executed before proceeding.

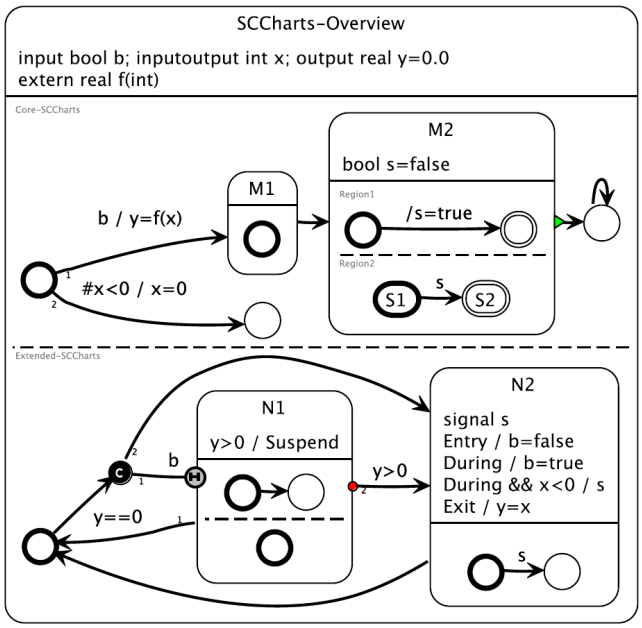


Figure 4.4. Syntactical Elements of SCCharts [vHMA+13b]

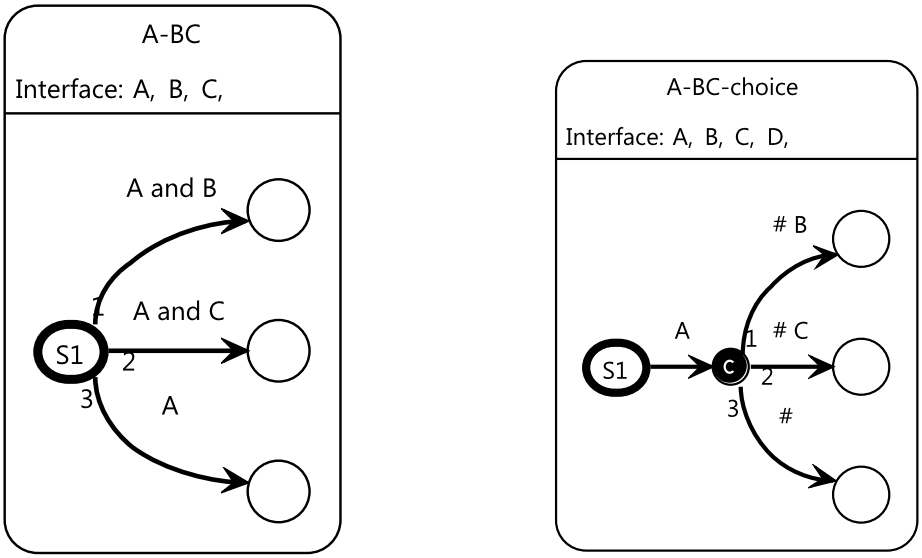
Extended SCCharts

The SCCharts variant enriched by more complex syntactical elements is called *Extended* SCCharts. These extensions aim to simplify the work of a modeller. They enable the user to express complex SCCharts in a compact way but do not alter the semantics of the Core SCCharts discussed earlier. Thus, all SCCharts elements can be considered syntactic sugar. Every Extended SCChart can be translated into a semantic equivalent Core SCChart. They extend the core variant by the following syntactical elements:

Signals: As mentioned in Chapter 4.1.1 Extended SCCharts reintroduce the concept of signals.

Connector: A connector is a transient state and must be exited immediately. To do so there must always be a valid outgoing transition. The connector is introduced to support the *Write-Things-Once (WTO)* principle and allows the combination of equivalent triggers and effects. A connector is also often referred to as *choice*.

Figure 4.5a depicts a statechart including a state S1 with three outgoing transitions. In each transition the signal A is tested. A semantically equivalent statechart that inherits the WTO principle is shown in Figure 4.5b. Here, A is only evaluated once and the three paths are connected to the choice node.



(a) Re-evalutation of A without WTO

(b) Single evalutation of A with WTO

Figure 4.5. The WTO Principle – Connectors

Strong Transition: Equivalent to SyncCharts, strong transitions pre-empt their preceding states and are executed at once, if their trigger evaluates to true. Thus, any kind of behaviour embedded in a state is aborted at once.

History Transition: A history transition returns to the state of the execution in which a macro state was pre-empted.

Suspend: As long as a connected condition is true the execution of a state and all its internal behaviour is suspended.

Entry Action: Entry actions are executed immediately when a state is entered. If a state is bypassed by a hierarchical pre-emption, the entry action is skipped.

During Action: During actions are executed in every tick as long as the execution persists in that state.

Exit Action: Exit actions take effect when their state is exited. They are executed if the state is pre-empted and the state was active before the pre-emption.

Pre: The *pre* operator returns the value of a signal or variable of the preceding macro tick.

Val: A *valued signal* holds a value in addition to its present state. The value is not limited to tick boundaries and persists as long as no new value is assigned. The *val* operator returns the value of a valued signal instead of its present state.

4.1.2 Sequential Constructiveness

This section recapitulates the goals of the Sequentially Constructive Model of Computation (SC MoC) introduced by von Hanxleden et al. in "Sequentially Constructive Concurrency - A conservative extension of the synchronous model of computation" [vHMA+13b].

The basic intention behind the SC MoC is to lift some of the restrictions of the classical synchronous MoC but to still rule out *race conditions* which may induce non-determinism. In contrary to the classical MoC it allows multiple accesses to the same variable as long as they are not concurrent conflicting writes. Furthermore, no restriction for sequential accesses are necessary and a write to some variable must be scheduled before any concurrent read.

Variable accesses

Variable accesses are *confluent* if the order in which they are executed does not matter. For instance, two concurrent assignments which assign the same value to a variable are confluent because no matter in which order the assignments are executed the variable will still hold the correct value.

Write accesses are further categorized in *absolute writes* and *relative writes*. *Relative writes* have the form $x = f(x, e)$ where f is so that such assignments are also confluent with each other. All non-relative writes are *absolute writes*. Furthermore, a *read* is a variable access which does not update the state of the variable.

To schedule different types of writes the sequential constructive approach organizes non-confluent concurrent variable accesses under a strict "initialise-update-read" protocol. In a tick instance, concurrent absolute writes must be executed at first as long as they are confluent with each other. If they are not, the program is not schedulable. Secondly, all concurrent relative writes can run, but must be scheduled after the absolute writes. Finally, all reads may proceed.

S-admissibility

Based on the variable access protocol introduced in the previous section, the following sequential constructive core definitions can be formulated.

1. An *S-admissible run* is an execution of a program which adheres to the "initialise-update-read" protocol with exception to confluent writes.
2. A program is *sequentially constructive*, or *S-constructive* in short, if there exists an S-admissible run and every S-admissible run generates the same deterministic output response.

The SC MoC in ABO

To illustrate the SC MoC Figure 4.3, mentioned in Section 4.1.1, is examined again. In ABO, only B has concurrent read/write accesses. The variable is written to in HandleA and read from in HandleB. Hence, the s-admissibility requires HandleA to be scheduled before HandleB. Since this is the only valid schedule for ABO, all executions will produce the same result and thus, ABO is sequentially constructive.

S-constructiveness Determination

As already pointed out by von Hanxleden et al. in "SCCharts - Sequentially Constructive Statecharts for Safety-Critical Applications" [vHMA+13a] the examination of S-constructiveness is of at least co-NP computational complexity. However, it can be conservatively approximated by testing whether a program is *acyclic sequentially constructive schedulable*, or *ASC-schedulable*.

3. A program is ASC-schedulable, if a static schedule exists which produces S-admissible runs.

Conclusively, a static schedule exists, if and only if a program does not contain concurrent, non-confluent writes and no cycles of concurrent write/read accesses.

4.1.3 The Sequentially Constructive Language

To handle and model SCCharts and their concurrent dependencies von Hanxleden et al. introduced the Sequentially Constructive Language (SCL) [vHMA+13b]. It is a minimal-

istic language adopted from C, Java and Esterel and is used to describe SCCharts in textual form.

SCL is a concurrent imperative language with shared variable communication. Hence, the underlying semantics are the same as for SCCharts variables can be both written and read from by concurrent threads. These reads and writes are collectively referred to as variable *accesses* [vHMA+13b].

SCL Instructions

SCL only consists out of seven different statements. Its program constructs have the following abstract syntax of statements

$$s \quad ::= \quad x = e \mid s_1 \ ; \ s_2 \mid \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_1 \mid l : s \mid \mathsf{goto}\ l \mid \mathsf{fork}\ s_1\ \mathsf{par}\ s_2\ \mathsf{join} \mid \mathsf{pause}$$

where x is a *variable*, e an *expression* and l is a *program label*. In detail the statement s comprises the following standard operations.

Assignments: An assignment $x = ex$ is a write to x variable access. The expressions ex in this SCL implementation use the underlying *SText* expression language of Yakindu. An assignment expression must not have any side effects. Note that ex may also hold references to other variables and the assignment as a whole may be a combination of read/write accesses.

Sequences: A sequence separates two consecutive statements. They are separated by a semicolon.

Conditionals: Similar to assignments, conditionals use *SText* to evaluate expressions. However, a conditional expression consists of read accesses only.

Labels: A label marks a specific spot in a program. It is represented by an identifier and closes with a colon.

Jumps: Jumps are initiated by the goto keyword. The target of a jump must be a statement identified by a label.

Parallel: The fork s par s join construct introduces concurrency to SCL. It forks off two threads and joins, when both threads are terminated.

Pause: A pause disables the actual thread until the next macro tick commences.

A *well-formed* SCL program is one

1. in which expressions and variable assignments are type correct,
2. that has no duplicate or missing program labels, and
3. has no goto jumps into or out of a parallel composition [vHMA+13b].

```
1 O = 0;
2 if I then
3     // true branch
4     O = 1;
5     goto exit
6 end;
7 // implicit else branch
8 O = 2
9 exit:
```

Listing 4.2. Implicit *else branch* in SCL

The implementation of SCL in this approach allows the creation of an arbitrary number of threads in a single parallel statement. More nested threads are spawned via additional par keywords in the parallel instruction.

Although the conditional instruction implements *else cases*, it has to be noticed, that implicit else branches are often emulated by a jump out of the *then branch* in which the normal control flow succeeding the conditional is posing as else branch. "Then branches" are also denoted as *true branches*. This behavior is illustrated in Listing 4.2. The output variable O is initialized with zero. If I is present, the program exits with O being one. Otherwise, O will be two.

As will be further described in the implementation in Chapter 5, this approach uses Xtext to build the SCL. Hence, a fully equipped editor including syntax highlighting and comments is available to model SCL. Figure 4.6 shows a simple transition example and Listing 4.3 depicts the corresponding SCL in the editor generated by Xtext.

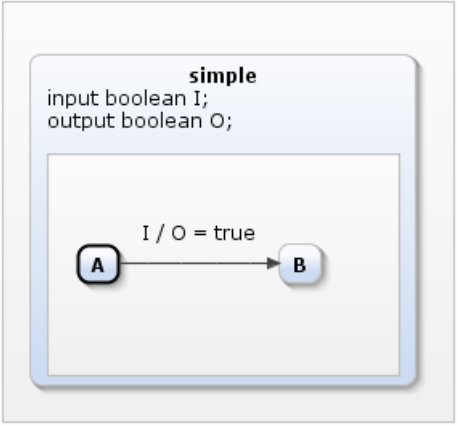


Figure 4.6. Simple Transition Example

```
1  module simple
2  input signal I;
3  output signal O;
4  {
5     __A:
6     pause;
7     if ! I then
8        goto __A
9     end;
10    O = true;
11    __B:
12    pause;
13    goto __B
14 }
```

Listing 4.3. Simple Transition in SCL

SCL Expressions

As already mentioned in the previous section SCL utilizes the expressions provided by the SCT, in particular *SText*. Even though SText comes with a powerful set of expression elements only a relatively small subset is needed to cover all aspects of SCL. Section 4.1.3 pointed out that expressions are used in assignments and conditionals. In *SText* these are built out of *REFEXP* and *ASSEXP*. Figure 4.7 and the following summary describe the *SText* expression elements used by the SCL.

Element reference expression: As the name suggests an REFEXP holds a reference to an element of a model. In the case of SCL this is usually a *variable declaration*. Thus, a REFEXP describes a single variable of the corresponding model in a given expression.

Assignment expression: An ASSEXP expresses an assignment to an element reference. It holds an operator type that specifies the kind of the assignment (equal, greater, etc.), an expression for the right hand side of the equation and the reference to the element in question.

LogicalRelationExpression: To test an equation for correctness a condition may use the Logical Relation Expression (RELEXP) to compare the left-hand side with the right-hand side. A relational operator defines the type of the comparison. However, in the boolean case an REFEXP is sufficient to check the reference for true.

Logical AND expression: A ANDEXP combines two expressions via a logical and operator usually depicted by a double ampersand (&&). In the context of this thesis the "and" concatenation of expressions will further be exemplified with the $\wedge$ operator.

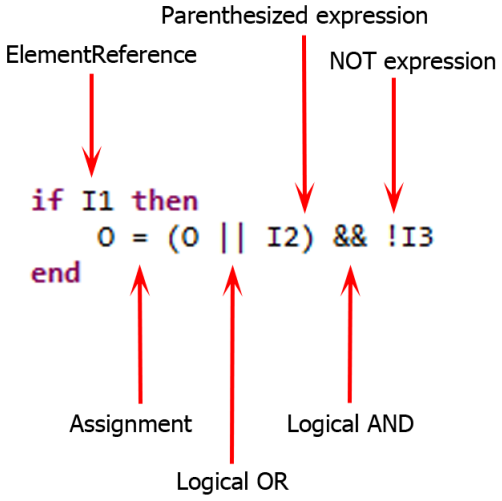


Figure 4.7. Illustrated SText Expressions

Logical OR expression: Similar to the ANDEXP a Logical OR Expression (OREXP) binds two expressions with a logical or. As usual in common the or operator is symbolized with double pipes (||). Conformable to the ANDEXP, "or" unions are depicted with $\vee$.

Negate expression: The Negate Expression (NOTEXP), an exclamation mark (!) in textual form, negates a given expression.

Parenthesized expression: To form complex expressions and determine the order of their evaluation an expression can be encapsulated in a PAREXP. The usual mathematical rules for parenthesis apply.

SCL Annotations

So far this section describes elements mandatory for SCL to represent the semantics of SCCharts. However, since SCL is meant to be an intermediate language the modeller may wish to include additional information for further transformations or visualization of the model.

To enrich the model with additional information for further processing the presented SCL implementation includes a lightweight *annotation* mechanism. Statements may contain an arbitrary number of annotations. While these do not alter the semantics of a given SCL program, they can be used to carry transformation and layouting information. An annotation is related to the statement it precedes.

An SCL annotation is initiated by an @ symbol followed by a mandatory keyword and a optional list of parameters.

$$annotation \quad ::= \quad @\mathsf{keyword}[: parameter[, parameter]^*]$$

The SCL Metamodel

All prerequisites being in place the SCL metamodel can be engineered. In general, a metamodel constitutes the elements of a model language and their relationships within a model. Hence, it is an abstract syntax of a language and is desired to be minimal with respect to redundancy. The SCL metamodel is depicted in Figure 4.8.

Program: Every SCL program is contained in a program and must have a name. It consists of a variable definition and a statement sequence

Variable Definition: Although the variable definition is not depicted in the figure, since it is inherited from SText, it is mandatory for SCL and elucidated here. A variable definition declares variables and their purpose. They have a type and may be marked as *input* or *output* variable to interact with the environment. Additionally variables may be initialized with a specific value of their type. The type is also inherited from SText, however, *extension points* for user-defined types exist.

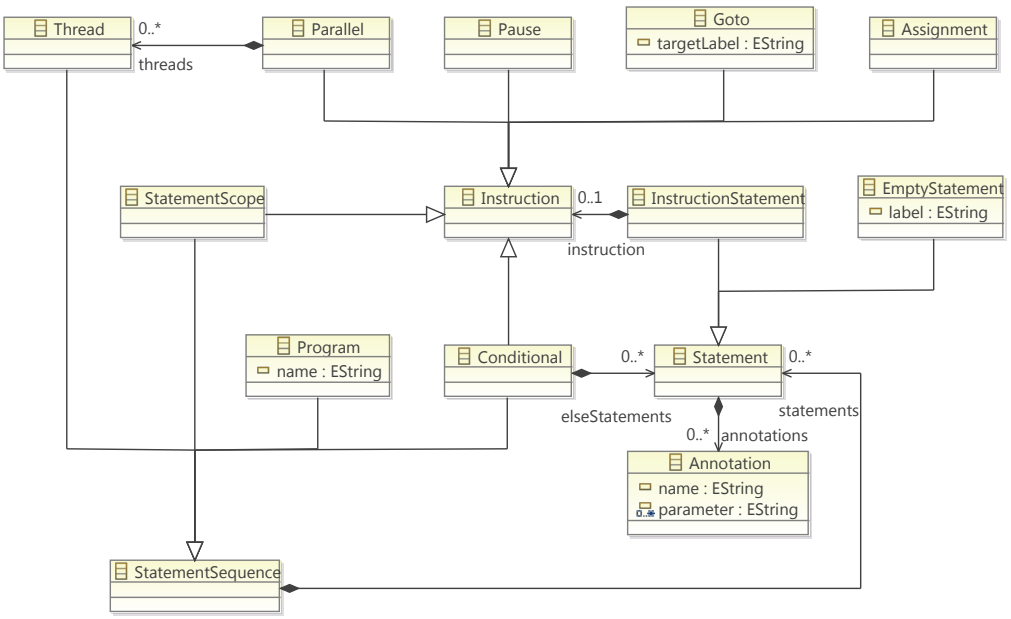


Figure 4.8. The SCL Metamodel

Statement Sequence: A list of statements is summarized in a statement sequence. According to the SCL sequence operator, they are separated by a semicolon. To simplify editing SCL, the last statement of a statement sequence may also finish with a semicolon. Semantically this implies a *no operation* as last instruction of a sequence.

Statements: There are two kinds of statements, Instruction and empty statements. The first type comprises all SCL instructions whereas the latter is introduced to hold labels because an SCL program is able to contain labels without instruction reference. Both are combined in the statement super class. As depicted in the previous section statements may also contain an arbitrary number of annotations.

Instruction: An instruction is an instance of one of the SCL instructions *assignment*, *conditional*, *goto*, *pause*, *parallel* or introduces a scope for internal variables, a statement scope. The conditional instruction contains a new statement sequence for the *true branch* and the parallel instruction comprises threads for the different concurrent control flows.

Thread: A thread is a statement sequence and models the execution order of a single control flow in the context of concurrent execution.

Statement Scope: It is possible to encapsulate sequences of statements in statement scopes to introduce new local variables or structure the control flow further.

Figure 4.9 depicts the automatically generated SCL code of ABO, Figure 4.3, and illustrates the discussed metamodel elements of the SCL.

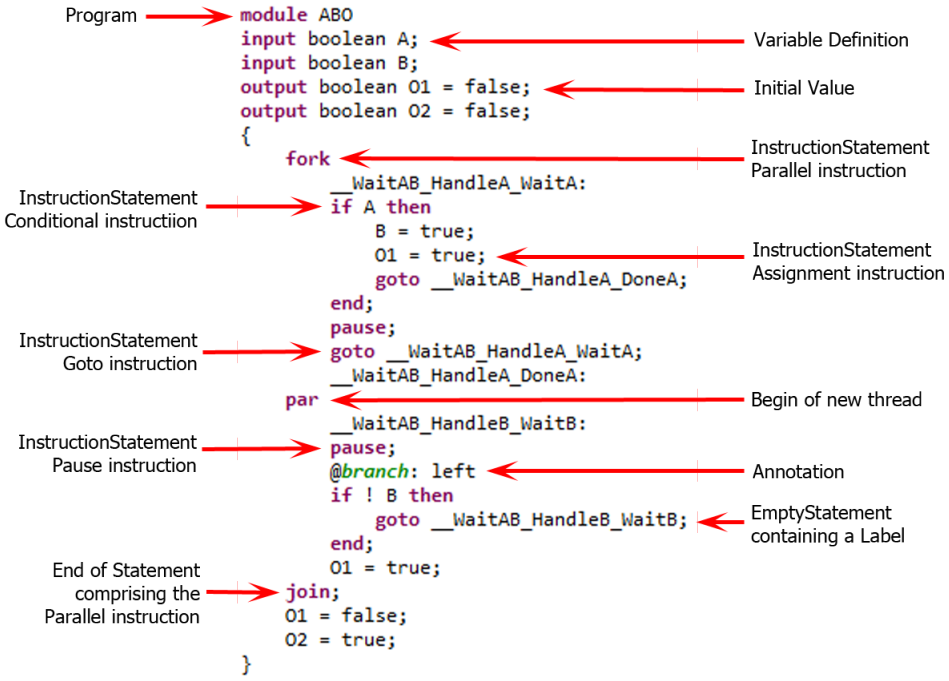


Figure 4.9. SCL Metamodel Illustration

4.1.4 The Sequentially Constructive Graph

The Sequentially Constructive Graph (SCG) is the graphical representation of the SCL. It is a labeled graph $G = (S, E)$, where

- ▷ the nodes S correspond to the statements of the SCL program and
- ▷ the edges E reflect the sequential execution control flow.

SCG Elements

Since SCG can be seen as graphical SCL representation all SCL statements must have a representative in the SCG. Figure 4.10 illustrates these elements.

Program: The program starts by the outer most entry node and terminates if the outer most exit node is reached

Assignments: Rectangular cornered figures represent SCL assignments.

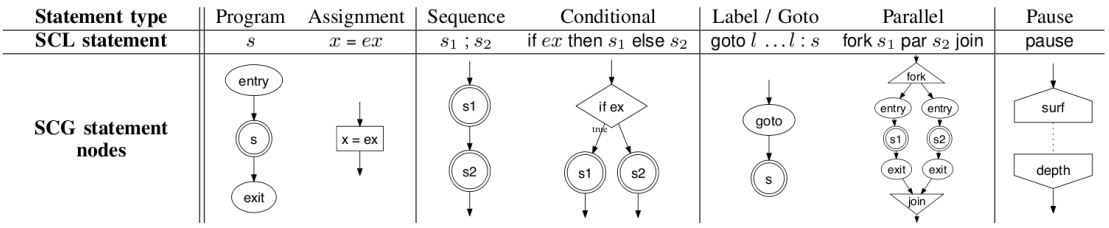


Figure 4.10. The SCG Figures [vHMA+13b]

Conditionals: Conditionals are depicted by a diamond figure. The true branch is indicated by the true keyword. Usually it is connected at the right side of the diamond but as shown in Figure 4.10 it can be annotated to switch the sides.

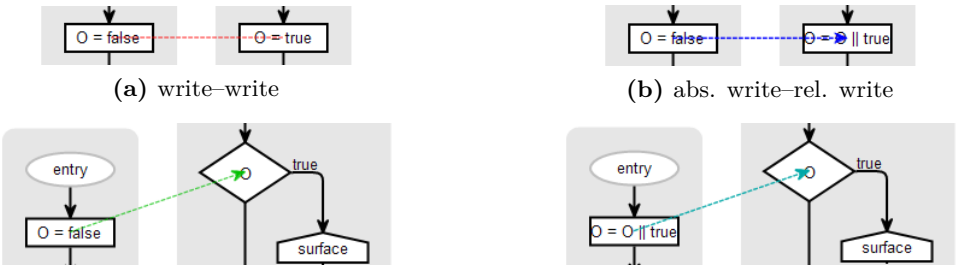
Parallel: The parallel statement is depicted by its corresponding fork and join nodes. The forked off threads also comprises entry and exit nodes.

Pause: A pause statement is divided in its *surface* and its *depth*. By default depth nodes are positioned at the top of the containing hierarchy to illustrate the tick start. However, for reasons of compactness they may be drawn *inline*.

Dependencies Visualization in the SCG

As elucidated in Section 4.1.2 there are several types of accesses and therefore equally many ways to illustrate a dependency are necessary. Figures 4.11a and 4.11b depict *write-write* dependencies. The left figure shows a dependency between two absolute writes. These are drawn red. The right one illustrates a dependency, a blue edge in the SCG, between an absolute and a relative write.

Even though there is no disparity in the handling of absolute or relative write-read dependencies, they are exemplified in a slightly different color to illustrate the write access. Figures 4.11c and 4.11d show the two kinds of write-read dependencies.



(a) write–write **(b)** abs. write–rel. write

(c) abs. write–read **(d)** rel. write–read

Figure 4.11. SCG Dependency Visualization

SCG Visualization Options

SCG comes with five global visualization options. Beside the possibility to draw the graph without further graphical information, it can be drawn with dependency edges and basic box illustration. Additionally, an option to switch off the display of hierarchy is available.

Furthermore it is possible to influence the arrangement of SCG elements by local annotations. Usually depth nodes of pause instructions are placed in the top layer of a diagram to illustrate the tick start. In some rare cases this is undesirable and can be suppressed by an inline annotation. Similarly *true branches* of conditionals are usually drawn on the right side of the diamond shape. A branch annotation may change the side of the branch. If desired, a layer annotation at a fork statement places entry and exit nodes in the outermost layers. The *node placement strategy* can be changed by a placement annotation at the beginning of the program code.

An example of a possible layout annotation case is depicted in Figure 4.12. It shows a simple SCL program comprising a pause and a subsequent assignment. As mentioned in Section 4.1.4 the depth of the pause is placed at the top of the diagram to exemplify the tick start. However, the modeler may annotate the pause to layout the model as illustrated in Figure 4.13, if desired.

```
1 module Main
2 output boolean O;
3 {
4     pause;
5     O = true;
6 }
```

Listing 4.4. SCL example – Pause

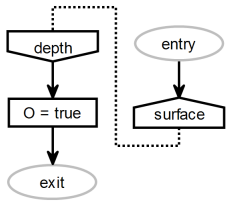


Figure 4.12. Pause visualization

```
1 module Main
2 output boolean O;
3 {
4     @inline
5     pause;
6     O = true;
7 }
```

Listing 4.5. SCL example – Annotation

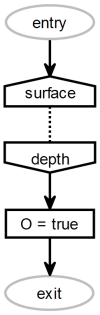


Figure 4.13. Annotated pause visualization

The complete ABO

As final example for the SCL and a corresponding SCG ABO, as depicted in Figure 4.3, was transformed to SCL and displayed with full visualization settings including dependency edges and BBs.

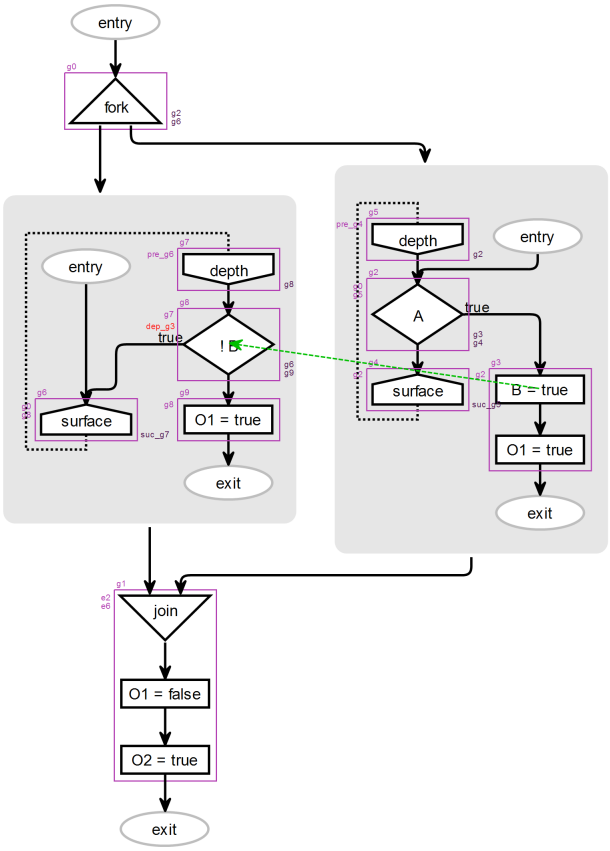


Figure 4.14. ABO with full visualization

In addition to Figure 4.3, Figure 4.14 illustrates the data dependency of B, denoted by the green dashed edge originating at the assignment in the right region. It also depicts the BBs as purple boxes encapsulating the SCG nodes. The concept of BBs is described in detail in the second part of this chapter.

The corresponding SCL code is shown in Listing 4.6. The fork introduces the first thread, lines 8-16, and the second thread, lines 18 to 24, is initiated by the par keyword. As elucidated in Section 4.1.4, each SCL statements relates to a figure in the SCG. The assignments correspond to the rectangles and the if instruction creates a diamond. Notice that the branch annotation in line 20 induces the control flow of the true branch to be drawn on the left side.

```
1  module ABO
2  input boolean A;
3  input boolean B;
4  output boolean O1 = false;
5  output boolean O2 = false;
6  {
7     fork
8        __WaitAB_HandleA_WaitA:
9        if A then
10          B = true;
11          O1 = true;
12          goto __WaitAB_HandleA_DoneA;
13       end;
14       pause;
15       goto __WaitAB_HandleA_WaitA;
16       __WaitAB_HandleA_DoneA:
17    par
18       __WaitAB_HandleB_WaitB:
19       pause;
20       @branch: left
21       if ! B then
22          goto __WaitAB_HandleB_WaitB;
23       end;
24       O1 = true;
25    join;
26    O1 = false;
27    O2 = true;
28 }
```

Listing 4.6. Complete SCL ABO

4.1.5 SCL Metamodel Extensions

The SCL metamodel provides the infrastructure to model and handle the mandatory data of an SCL program. However, to simplify the usage of SCL models, several Xtend model querying and transformation extensions have been implemented.

The SCL Factory Extension: The SCL Factory Extension subsumes all needed model factories. This includes factories for the *SGraph*, *SText*, *SyncGraph*, *SyncText* and *SCL*. Since this extension is the base of the other extensions it also contains methods for debugging.

The SCL Create Extension: The create extension aims to ease the work necessary to create new model constructs. It provides simple methods for SCL element creation as well as generating more complex components such as assignments which are built out of SCChart transition effects and need references to variable declarations.

The SCL Naming Extension: To simplify the identification of model elements the SCL Naming Extension provides functions to set unique Identifiers (IDs) and generate names for regions and states. The names are generated recursively throughout the hierarchy and are separated by an underscore. If a region or state does not have a name, the *hash code* of the model element is used instead.

The SCL Ordering Extension: The methods provided by the ordering extensions can be used to customize Xtend's sort extension. For instance, compareSCLStateOrder sorts states according to their type.

The SCL Statement Extension: The statement extension comprises convenient methods to avoid excessive instance checking and type casting. Although Xtend provides simplified constructs to handle these, they can be exhaustive over time. Methods such as getInstruction, hasAnnotation and isEmptyStatement are supported.

The SCL Statement Sequence Extension: To help the programmer to travel through the instructions of an SCL program the statement sequence extension provides methods to retrieve the actual program, thread or statement sequence. It also grants access to sequence compare functions and finds least common ancestors. Additionally, several next and previous methods allow travelling through the statements of a model respecting hierarchy and goto jumps.

The SCL Goto Extension: Since a goto targets a label and not an instruction directly, the goto extension does simplify the effort to find corresponding statements. It provides methods to find instruction statements targeted by a goto and also retrieves all incoming jumps of a statement. For instance, a target instruction of a goto jump can easily be determined by using goto.getTargetStatement?.getInstructionStatement. The question mark operator is provided by Xtend and checks if the given object is valid to avoid *null pointer exceptions*.

The SCL Expression Extension: The expression extension aids in the creation of new and the transformation of present expressions. It contains methods such as createAssignmentExpression, createAndExpression and negate. If the negate method is executed on an already negated expression, it removes the negation. Otherwise, a new negation is created and parenthesized if necessary.

The SCL Dependency Extension: This extension package assists in the detection and characterization of dependencies. It examines SCL assignments and conditionals and provides checks for *absolute* and *relative* writes, *confluent* tests and returns lists of concurrent dependencies and their kind.

The SCL Basic Block Extension: The Basic Block extension divides the SCL program in distinct blocks of statements. Essentially, they illustrate the connection between groups of instructions. A BB can be *active* or *inactive* in any tick. Its activity state depends on certain *guards*, activation dependencies to other BBs or the initial *GO* signal. Hence, every basic block may have successor dependencies which pose as guards to other blocks. Additionally, data dependencies of instruction inside a BB may impose ordering constraints to the program in concurrent threads. If these constraints are cyclic and cannot be resolved, the program is rejected.

4.1.6 Sequential Sequentially Constructive Language

Sequential SCL is a subset of SCL omitting the pause and the parallel instructions. Every statically schedulable SCL program can be translated into a concurrency-free, sequential SCL program. A sequential SCL program is meant to serve as a generic function which is called in every tick. This *tick function* comprises a netlist-like structure and evaluates the statically calculated *guards* to determine the control flow in every macro tick. Due to the omitting of the pause instruction, guards of blocks that depend on activation states of the previous tick must be saved in additional guard variables. These are suffixed with _pre.

Since the calculation of these guards is elucidated in detail in Section 4.2 the sequential SCL program shown in Listing 4.7 is meant as simple example to understand the previous paragraph. It depicts the transformed pause SCG, illustrated in Figure 4.12, and shows the guard calculation in the assignments. The time consumption of the register is semantically embedded in the g0_pre guard.

```
1  module tickfunction
2  output boolean O;
3  boolean GO;
4  boolean g0;
5  boolean g0_pre;
6  boolean g1;
7  {
8     g0 = GO;
9     g1 = g0_pre;
10    if g1 then
11       O = true;
12    end;
13    g0_pre = g0;
14 }
```

Listing 4.7. Tick Function Example – Pause

Once the sequential SCL is created its generic tick function can be used to further generate code for hardware or software synthesis.

4.1.7 Normalized Core SCCharts

While working on this topic a new dialect of SCCharts evolved. Although it is not used in the implementation of the approach presented in this thesis, it is discussed in this section for the sake of completeness. The newly developed Normalized Core SCCharts (NCSC) are a variant of Core SCCharts and aim to omit the intermediate SCL translation step. Besides parallelism, they only permit three different kinds of state connectivity, depicted in Figure 4.15, but do not differ from Core SCCharts semantically. Every connectivity type corresponds with an element of the SCG. Hence, only a MTM transformation is needed to display a corresponding SCG. Each Core SCChart can be expressed by this normalized variant and therefore, Extended SCCharts are covered as well. Connectors

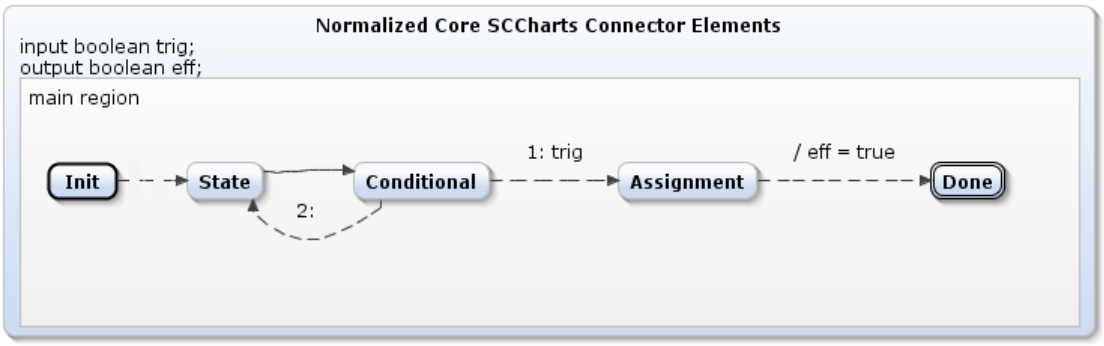


Figure 4.15. Normalized Core SCCharts Connector Elements

ease the use of transient states and facilitate the WTO principle. The different types are described as follows.

State: A state has a single outgoing transition which consumes time. The transition is seen as continuous line outgoing from state State in Figure 4.15. As before, time consumption in the SCG is illustrated by a dotted edge between the surface and depth node.

Conditional Connector: A conditional connector stands for a triggered transition and is depicted as conditional node in the SCG. The SCL equivalent is the if statement. If no trigger evaluate to true, the control immediately returns to the calling state. Thus, the state Conditional in Figure 4.15 is transient.

Assignment Connector: The assignment connector executes transition effects. It is also a strictly transient state and represents an SCL/SCG assignment.

As usual, parallelism is still modeled by concurrent regions.

The connector types may be translated directly to corresponding SCG elements without intermediate steps, depicted in Figure 4.16. As a state consumes time, it is represented by a pause in the SCG without intermediate step. The conditional connector comprising the trigger trig is related to the conditional diamond figure including the trig expression. Subsequently, any assignment connector is translated to an SCG assignment node. It is depicted by the eff = true expression in the figure. Any parallelism in the NCSC would result in concurrent compartments in the SCG visualization.

This direct transformation can be done transiently, thus making changes to an Extended SCChart visible at once in control flow form. Furthermore, it is possible to express all syntactical sugar of Extended SCCharts with these four essential elements of Core SCCharts.

This direct transformation can be done transiently, thus making changes to an Extended SCChart visible at once in control flow form. Furthermore, it is possible to express all syntactical sugar of Extended SCCharts with these 4 essential elements of Core SCCharts.

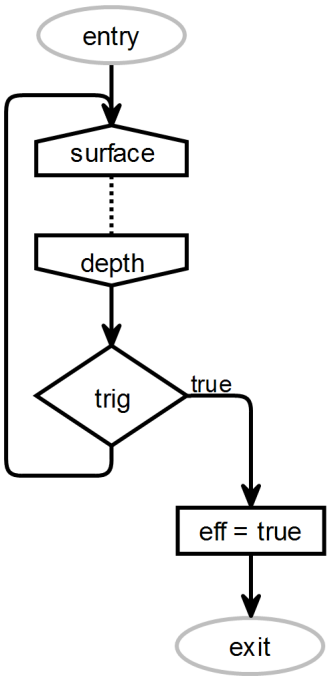
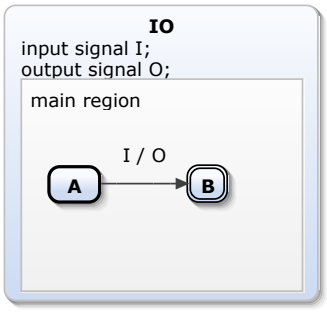


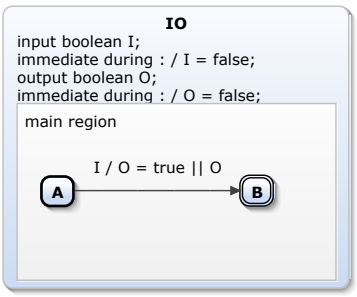
Figure 4.16. Normalized Core SCCharts Connector Types in the SCG

4.2 Sequential Constructiveness Transformations

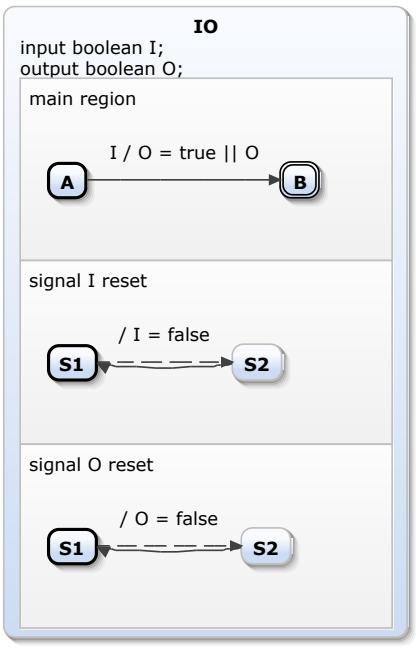
As described in the introduction of this chapter a stack of transformations must be processed to obtain the desired sequential tick function. Starting from Extended SCCharts the corresponding Core SCCharts are generated to transform the syntactical sugar, described in Section 4.1.1, to its semantically equivalent core variant. Secondly, Core SCCharts are translated to SCL code and their related SCGs are synthesized to gain a model and a visualization of the control flow as depicted in Section 4.1.3 and Section 4.1.4. Subsequently, SCL is analysed with respect to schedulability and finally converted to sequential SCL discussed in Section 4.1.6. Therefore, the generated sequential tick function will no longer comprise concurrent threads or registers. The next sections will exemplify each step of the transformation chain.



(a) Extended SCChart IO



(b) Intermediate IO



(c) Core SCCharts IO

Figure 4.17. Example of Transforming Extended SCCharts to Core SCCharts

4.2.1 Extended SCCharts Expansion

As already mentioned in Section 4.1.1, Extended SCCharts enrich the syntax of Core SCCharts by more complex elements to allow the modeler to build equally sophisticated models with respect to Core SCCharts in a more compact way.

In principle each added syntactical extension can be translated to a core variant with one or more MTM transformations. The Extended SCChart is searched for occurrences of the element in question and then transformed to a more expanded chart. However, this must not be a core variant directly. In most cases the final Core SCChart is reached over a series of consecutive transformations.

As example for an Extended SCCharts expansion chart IO, Figure 4.17a, shall be transformed to a core equivalent. It is not a Core SCChart since it comprises signals which belong to the Extended SCCharts supplement. In a first step IO is transformed to an intermediate variant depicted in Figure 4.17b. Signals can be emulated with boolean variable, input boolean I and output boolean O in the figure. However, the present status of a signal is reset in each macro tick. This is modelled by the immediate during actions. Therefore, this SCChart is still an extended variant since during actions belong to the repertoire of Extended SCCharts.

Hence, the intermediate SCChart IO must be transformed further. In a second step,

its during actions are resolved to concurrent regions. This transformation step is depicted in Figure 4.17c. Since this SCChart only comprises core elements, it is valid Core SCChart and thus, the signal transformation is completed.

Since it is not the main focus of this thesis, interested readers are referred to "SCCharts - Sequentially Constructive Statecharts for Safety-Critical Applications" [vHMA+13a] for more detailed information about Extended SCCharts Expansion.

4.2.2 Core SCChart to SCL Transformation

One of the challenges of this code generation approach is the efficient transformation of SCCharts models to SCL code. The created code must be correct and is desired to be as compact as possible. Therefore, the transformation of Core SCCharts to SCL proceeds in two stages. Firstly, SCCharts are translated according to a relatively straight-forward pattern. Secondly, the generated SCL code is optimized where possible.

Straightforward SCL Transformation

As mentioned in Section 3.3 every SCChart is stored in an SGraph *Statechart*. The statechart comprises the *main state* which may contain one or more regions. The transformation is implemented in Xtend and travels recursively through all regions and states.

In the region transformation all containing states are ordered according to their type. Initial states are placed at the beginning of the state list and final states come last. Eventually, every state instance invokes the transformation method for states. Lastly, applicable optimizations are called.

The transformation pattern for states is illustrated in Figure 4.18. It is similar to the approach presented by Amende [Ame10] discussed in Section 2.1, since each SCCharts element strictly corresponds to distinctive code patterns.

At first a state gets a name respecting hierarchy and is identified by a label. If it is a final state, only its label is mandatory in the SCL code and the translation finishes immediately. If it is not a final but a composite state, a parallel statement is created and the transformation for regions is called again. Eventually any existing normal termination is translated. If it is not a composite and not a final state, all outgoing transitions of the state must be processed.

In the transition translation all outgoing transitions are checked for their immediate flag. If a transition is immediate, it has to be processed before any normal transition and according to their priority. Additionally any transition without trigger is marked. This kind of transition is also called *default transition* since it activates if no other trigger evaluates to true. After all immediate transitions are processed a pause statement is included and subsequently the transformation handles all outgoing transitions. This also includes all immediate processed before since the priorities may be different in connection with non-immediate transitions. Similar to the first transition transformation

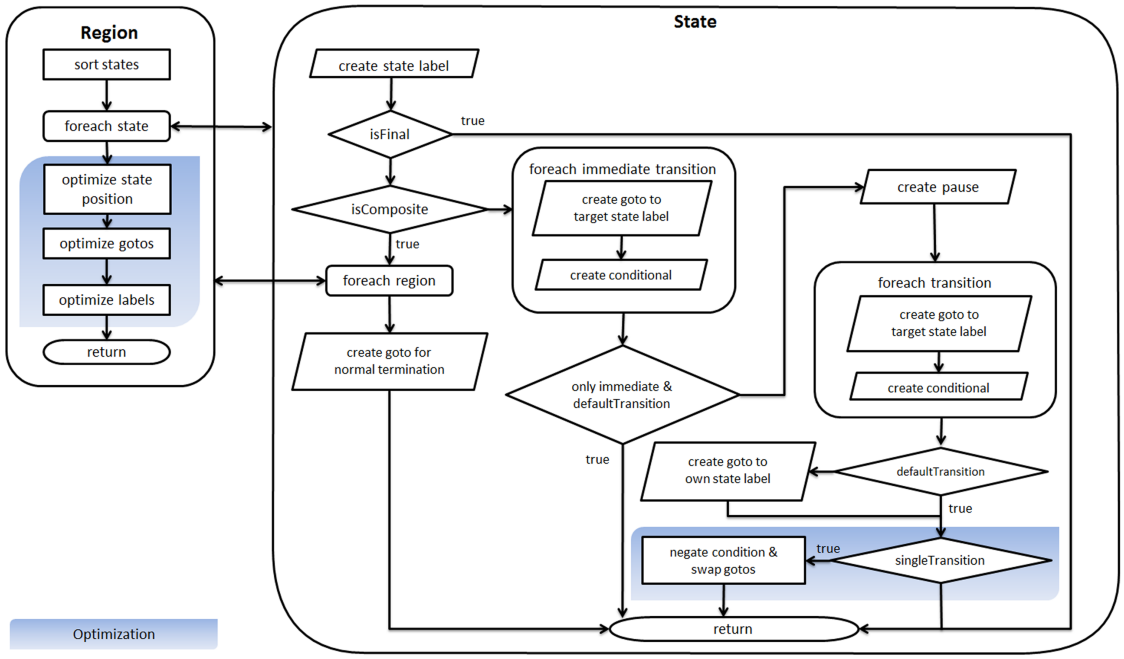


Figure 4.18. State Transformation Pattern

it is checked whether or not a default transition is present. The second translation of outgoing transitions can be skipped when no non-immediate transitions are present.

Transitions are translated in a manner straightforwardly. Every trigger corresponds to a conditional statement and every effect complies to an assignment in the SCL code. The code of a transition assignment is executed in the statement sequence of the corresponding conditional. If the trigger is omitted, the assignment belongs to a default transition and is executed in a tick instance if no trigger evaluates to true. Subsequently the transition to another state is translated to a goto jump. Therefore, if the transition only comprises of a trigger and no effect, the goto statement is the only statement of the corresponding conditional.

Eventually, if no default transition was found, a *self-loop*, a goto jump to the state itself, is created. If all transitions of the state are immediate and a default transition exists, the state is a transient state and does not consume any time. Therefore, no pause statement is needed in the SCL code.

ABO in SCL

Listing 4.8 exemplifies the SCL code generation of ABO, Figure 4.3. The main state ABO comprises the states WaitAB and GotAB. These are indicated by the labels __WaitAB and __GotAB in the SCL code. The state WaitAB consists out of two regions, HandleA and HandeB. Since they are modelling concurrency a parallel statement with two concurrent

threads is generated. It is depicted by the fork-par-join construct in the listing.

HandleA includes the states WaitA and DoneA which are connected via a immediate transition. The trigger of the transition listens for the input A. As effect B and O1 are set to true. In the SCL code WaitA is initiated by the recursively created label __WaitAB_-HandleA_WaitA. The following if A then instruction models the outgoing immediate transition and includes the effects in its statement sequence. The transition traversal is done by the goto jump to DoneA. Subsequently, the pause statement is included. Since there are no further outgoing transitions, the implicit self-loop to WaitA is created. As last statement the label of the final state DoneA marks the end of this thread.

In HandleB, as the transition from WaitB to DoneB is not immediate, the if B then construct is placed after the pause statement. Similar to WaitB a self-loop is introduced at the end of WaitB. The thread is finished, if label __WaitAB_HandleB_WaitB is reached. Eventually both regions reach their final state. Since the normal termination has no trigger, its effects are translated directly. As last instruction the goto jump to __GotAB is executed.

```
1  module ABO
2  input output boolean A;
3  input output boolean B;
4  output boolean O1 = false;
5  output boolean O2 = false;
6  {
7     __WaitAB:
8     fork
9        __WaitAB_HandleA_WaitA:
10       if A then
11          B = true;
12          O1 = true;
13          goto __WaitAB_HandleA_DoneA;
14       end;
15       pause;
16       goto __WaitAB_HandleA_WaitA;
17       __WaitAB_HandleA_DoneA:
18    par
19       __WaitAB_HandleB_WaitB:
20       pause;
21       if B then
22          O1 = true;
23          goto __WaitAB_HandleB_DoneB;
24       end;
25       goto __WaitAB_HandleB_WaitB;
26       __WaitAB_HandleB_DoneB:
27    join;
28    O1 = false;
29    O2 = true;
30    goto __GotAB;
31    __GotAB:
32 }
```

Listing 4.8. Unoptimized ABO SCL

4.2.3 SCL Code Optimization

So far the SCL generation is relatively naive. Several optimizations are in place to improve the overview and compactness of a SCL program without altering its semantics.

Goto Optimization: The first optimization searches for goto instructions with jumps to labels that directly follow that goto. These instructions are superfluous since the sequential control flow will already proceed in that direction.

Listing 4.9 depicts the last five lines of the unoptimized ABO. The goto instruction in line 30 precedes its target label __GotAB promptly. Thus, it can be eliminated since the control flow will proceed to line 31 anyhow as illustrated in Listing 4.10.

```
28      O1 = false;
29      O2 = true;
30      goto __GotAB;
31      __GotAB:
32  }
```

Listing 4.9. Goto Optimization Example in ABO before Optimization

```
28      O1 = false;
29      O2 = true;
30
31      __GotAB:
32  }
```

Listing 4.10. Goto Optimization Example in ABO after Optimization

Label Optimization: Secondly, every not referenced label can be deleted, because there are no jumps to it and is therefore redundant.

Now that the superfluous goto in line 30 was removed, shown in Listing 4.11, the last label __GotAB is not referenced any more and can also be deleted as illustrated in Listing 4.12.

```
28      O1 = false;
29      O2 = true;
30
31      __GotAB:
32  }
```

Listing 4.11. Label Optimization Example in ABO before Optimization

```
28      O1 = false;
29      O2 = true;
30
31
32  }
```

Listing 4.12. Label Optimization Example in ABO after Optimization

Self-loop Optimization: If a state has only one outgoing non-default transition, the expression of the trigger may be negated to invert the self-loop and effect code. Thus, the effect will be executed, when the control flow proceeds to the next state and no goto and label are mandatory to execute this transition.

This is exemplified in Listing 4.13 and Listing 4.14 in the code pattern of the state WaitB. Since it has only one outgoing transition, the expression can be negated to exploit the natural control flow of the program. Therefore, the second goto instruction in line 25 also precedes its target label and thus, both can be eliminated also.

```
28        if B then
29          O1 = true;
30         goto __WaitAB_HandleB_DoneB;
31        end;
32        goto __WaitAB_HandleB_WaitB;
33        __WaitAB_HandleB_DoneB:
```

Listing 4.13. Self-loop Optimization Example in ABO before Optimization

```
28        if !B then
29           goto __WaitAB_HandleB_WaitB;
30        end;
31        O1 = true;
32        goto __WaitAB_HandleB_DoneB;
33        __WaitAB_HandleB_DoneB:
```

Listing 4.14. Self-loop Optimization Example in ABO after Optimization

State Ordering Optimization: As mentioned in Section 4.2.2 all states in a region are ordered according to their initial or final type. The order of the remaining states directly influences the creation of goto jumps because the goto optimization will remove unnecessary goto instructions of subsequent states. This optimization sorts states according to their transitions in order to further facilitate the goto and label optimization.

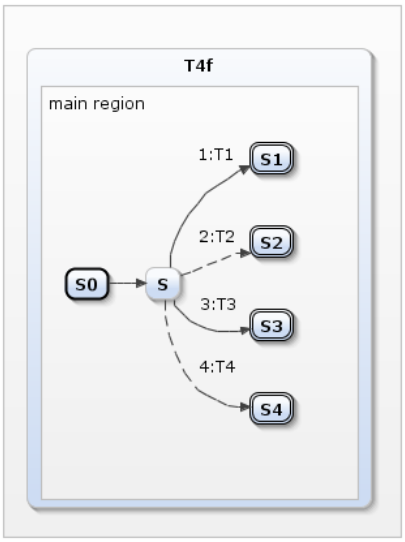
Duplicate Transition Optimization (DTO): Due to the blending of priorities of immediate and non-immediate transitions, all immediate transitions must be processed twice. Firstly, they must be checked before the pause instruction and then again in conjunction with the normal transitions afterwards. Therefore, two if instructions are created for every immediate transition. Depending on the transition priorities used in the statechart it may occur that a transition is checked twice consecutively in one tick instance because the goto at the end of the state loops back to the beginning of the state and checks the transitions again. This optimization removes all unnecessary if instructions in the depth of a state if the transitions will be evaluated again in the surface and are in the right order.

WTO Transition Optimization (WTOTO): With regard to their priority even with the DTO enabled, an immediate transition may introduce two if instructions in the SCL code. To facilitate the WTO principle this optimization only creates one if instruction per transition but must introduce a local variable to do so. This boolean variable marks the depth of a state and is added to the triggers of the non-immediate transitions. It is initialized with false and set to true after to the pause instruction. Thus, subsequent transition checks include non-immediate transitions.

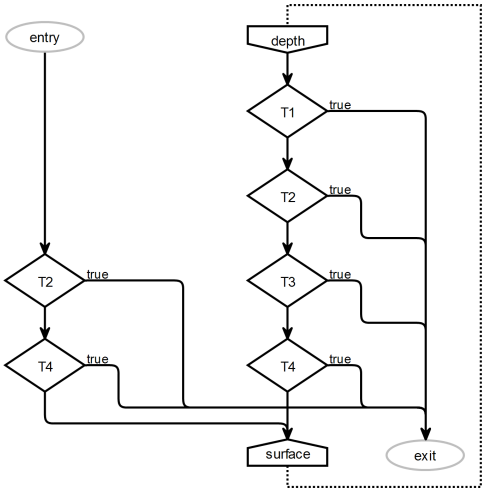
Depending on which transition optimization is chosen the Figure 4.19a translates to an SCL program with an SCG corresponding to one illustrated in Figures 4.19b to 4.19d. The final states have been combined to one exit node to display the SCG in a more compact way.

Figure 4.19b depicts an unoptimized translation with duplicated conditionals T2 and T4. In Figure 4.19c one T4 conditional is eliminated by the DTO. The second conditional is not removed because the priorities forbid it. In the last example, Figure 4.19d, the WTOTO introduces a new variable dep to handle the depth of the state. Thus,

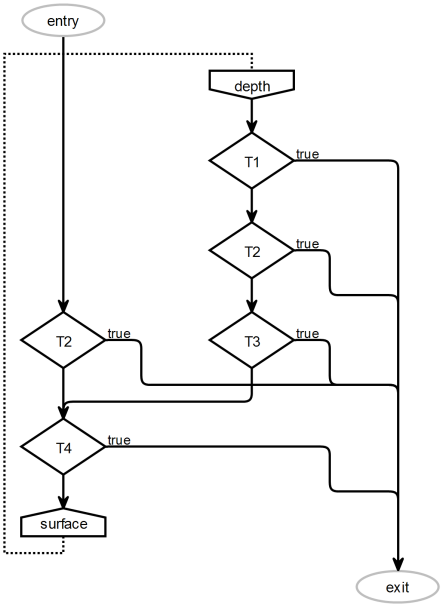
all duplicated transitions can be removed. dep is initialized with false and set to true after the *depth* of the state is reached.



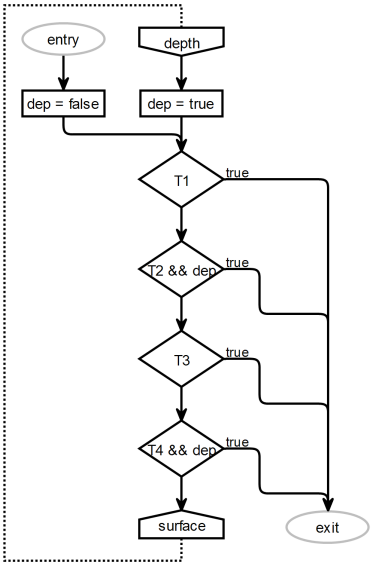
(a) T4f SCChart



(b) Naive Translation



(c) DTO Translation



(d) WTOTO Translation

Figure 4.19. T4f Transition Translation Example

Listing 4.15 depicts the optimized SCL of ABO. The self-loop optimization improved the WaitB code generation and all superfluous goto instructions and labels are removed.

```
1  module ABO
2  input output boolean A;
3  input output boolean B;
4  output boolean O1 = false;
5  output boolean O2 = false;
6  {
7     fork
8        __WaitAB_HandleA_WaitA:
9        if A then
10          B = true;
11          O1 = true;
12          goto __WaitAB_HandleA_DoneA;
13       end;
14       pause;
15       goto __WaitAB_HandleA_WaitA;
16       __WaitAB_HandleA_DoneA:
17    par
18       __WaitAB_HandleB_WaitB:
19       pause;
20       if !B then
21          goto __WaitAB_HandleB_WaitB;
22       end;
23       O1 = true;
24    join;
25    O1 = false;
26    O2 = true;
27 }
```

Listing 4.15. Optimized SCL of ABO

4.2.4 SCG Synthesis

The control flow of an SCL program can be visualized graphically. Due to the fact that the SCL metamodel already represents the sequential execution flows of the corresponding program, the visualization can be created transiently on the SCL without further transformations. This is useful to get an overview of the general control flow and detect scheduling problems due to concurrent variable accesses discussed in Section 4.1.2.

The code analyses, presented in the succeeding Section 4.2.5 and Section 4.2.6, will use the possibilities provided by the SCG visualization to hint at potential problems during the code generation.

The SCG is displayed via KLighD and triggered over a KIVi combination. Whenever the SCL model changes the synthesis is invoked automatically and the corresponding SCG is shown. This behavior can be toggled in the KIVi control menu.

The creation of graph figures is unsophisticated. For every SCL instruction an analogous synthesis method exists. It creates the SCG figure and returns edge connection points for subsequent figures. To connect the edges ports are used. Eventually, KLay is invoked to layout the generated graph. Unlike to most figure methods the synthesis for the parallel

and pause instructions create two figures. In the case of parallel a figure for fork and join each are generated. Similarly a surface and a depth node are composed for pause. A detailed description follows in Chapter 5. Figure 4.20 illustrates the final version of ABO in its SCG control flow form. Each program starts with an entry and closes with an exit node. The fork-par-join instruction results in the creation of a fork node, two thread regions, denoted as gray compartments and a corresponding join node. In the figure the left thread depicts HandleB and the right one represents HandleA. Each thread also comprises an entry and an exit point. As signal A is tested for presence immediately, the control flow proceeds to the conditional diamond at once. Depending on its evaluation the path to the pause or the assignments preceding the exit node is taken. In HandleB, the status of B is checked after the pause as illustrated by the control flow edges. Eventually, both threads terminate and the control flows of the concurrent regions merge in the join. Finally, the two O assignments are executed and the program terminates.

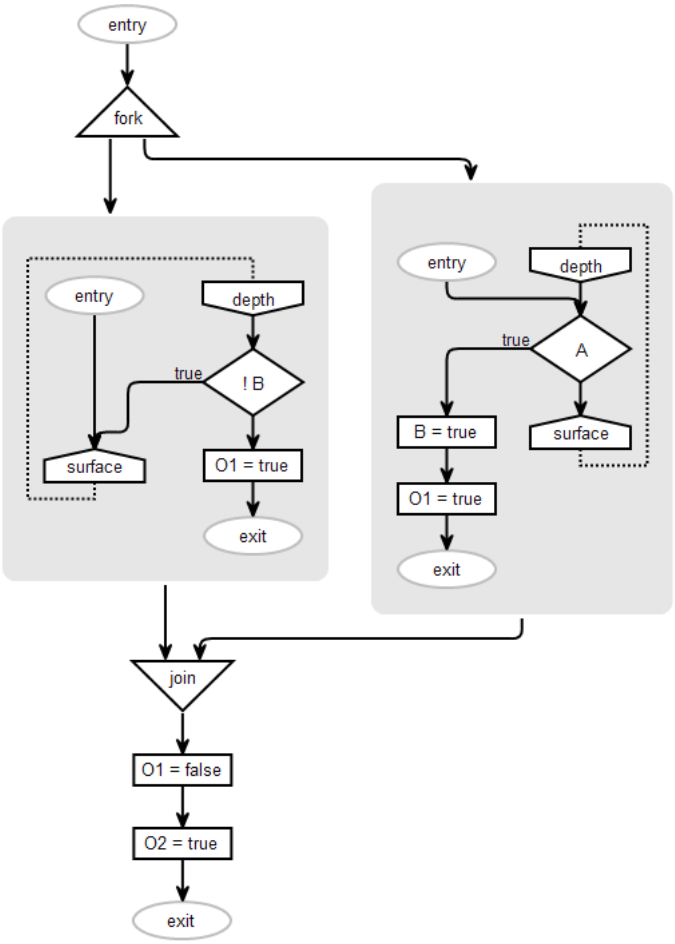


Figure 4.20. The SCG of ABO

4.2.5 Dependency Analysis

To gather information about the schedulability of an SCL program concurrent dependencies must be evaluated. As discussed in Section 4.1.2 programs are not schedulable, if they comprise any conflicting variable accesses. Therefore, potentially problematic accesses must be found and categorized with respect to their access type.

This section defines the term of concurrency with regard to SCL programs and describes the domain in which variable accesses are treated as concurrent.

Concurrency

The set of threads of an SCG G with nodes N is denoted T. Every thread $t \in T$ is associated with unique entry and exit nodes $t.entry$, $t.exit \in N$. Every $n \in N$ belongs to a thread $thread(n)$, defined as the immediately enclosing thread $t \in T$ such that there is a control flow path to n that originates in $t.entry$. Let $fork(t)$ be the fork node that immediately precedes $t.entry$. Every thread that is not the root thread has an immediate *parent thread* $p(t)$, defined as $thread(fork(t))$. A set of *ancestors threads* $p^*(t)$ is recursively defined as a set of t, $p(t)$, $p(p(t))$, ..., program root.

> ▷ Two threads $t_1, t_2 \in T$ are *concurrent*, if and only if there exists $t_1' \in p^*(t_1)$, $t_2' \in p^*(t_2)$, with $t_1' \neq t_2'$, which share a common fork node $fork(t_1') = fork(t_2')$. This fork node is referred to as the *least common ancestor fork*, $lcaf(t_1, t_2)$ [vHMA+13b].

Detecting Concurrent Dependencies

As pointed out in Section 4.1.5 most of the work necessary to find and categorize concurrent dependencies is done by the SCL dependency extension. It examines the SCL model and comprises methods for variable access detection and categorization of any given access. Technical details about the dependency analysis are discussed further in Chapter 5.

However, it is important to understand that according to the definition of concurrency two accesses in different threads are only concurrent, if and only if they share a least common ancestor fork. Only if they are concurrent, they must be tested for potential conflicting accesses. Otherwise, two writes to the same variable in two different threads in the same tick instance will be detected as conflict even though one thread is executed subsequent to the other and thus, constitutes no conflict at all.

Figure 4.21 illustrates least common ancestor forks and the dependencies between multiple nested concurrent threads. The first fork creates two threads. The thread shown on the left t_l consecutively forks two threads two times t_{l_1}, t_{l_2} and the one on the right t_r assigns an integer to x only. t_{l_1} concurrently assigns two values to y and t_{l_2} assigns integers to x and y.

As depicted by the first red dependency edge, the two y assignments in t_{l_1} impose a concurrent write access. Nevertheless, the y assignment in t_{l_2} is not concurrent to the y

assignments in t_{l_1} since they do not share a least common ancestor fork node. However, since t_r is not nested inside of t_l, t_r shares a least common ancestor node with t_{l_2} and thus comprises a write dependency to a statement of t_{l_2}.

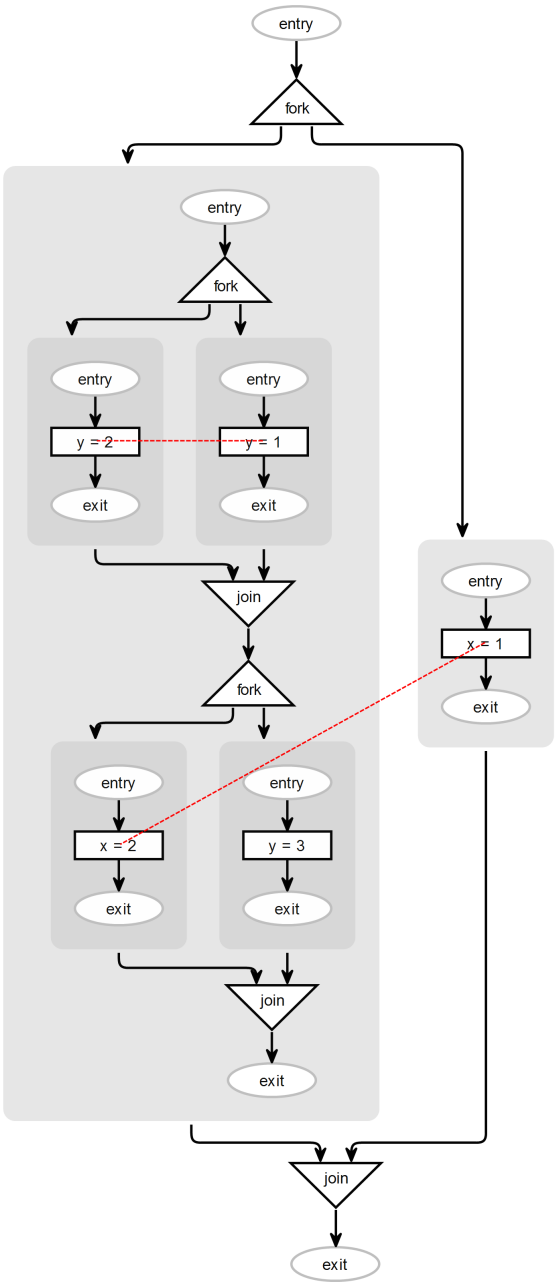


Figure 4.21. Least Common Ancestor Fork Example

4.2.6 Basic Block Analysis

A Basic Block (BB) is an amalgamation of SCL statements. It comprises a list of instructions in any SCL control flow that can be executed as a single block without rescheduling. Conservatively, it is possible to build BBs out of single statements. However, as it is desirable to connect as many instructions as possible in a single BB before a reschedule may be necessary, the following rules apply.

- ▻ A BB begins with the statement at the beginning of a thread or if the SCG representation of that statement has two or more incoming edges. An incoming edge may be a control flow or a dependency edge.
- ▻ A BB ends with a statement that forks the SCG control flow and hence, has two or more outgoing control flow edges. The last instruction of a thread may also be the closure of a BB.
- ▻ BBs are split at pause statements.
- ▻ SCG fork node close a BB, whereas join nodes start a new one.
- ▻ Any statement of a given program can only be included in one BB at any time.

Splitting up SCL Constructs

To avoid unnecessary consistency constraints and data dependencies no extra metamodels for the SCG or BBs were introduced. All evaluations are done transiently on the SCL model via Xtend extensions. Hence, a BB is identified by any of its statements. In general, the first one is used. However, as specified in the previous section, a block is separated by pause and parallel instructions, even though they are comprised out of a single statement in SCL. This would violate the rule that a statement can only be present in one BB at any time since it would be the last instruction of the preceding BB and the first one in its successor. To avoid excessive surface and depth handling, the Basic Block extension transiently splits every pause in a PauseSurface and a PauseDepth object and analogously every parallel in a ParallelFork and a ParallelJoin. Hence, in the context of BBs the abstract syntax of the SCL statements is extended by these instruction splits and a statement uniquely defines a BB again.

Basic Block Definition

In every tick instance a BB may be *active* or *inactive*. The activity state of a BB derives from its dependencies to previous BBs and is called the *guard* of the BB. The guard of the first block in an SCL program depends on the *GO* start signal of the environment usually emitted at the initialization or reset of the program. Outgoing control flows of a BB may pose as *guard expressions*, or *activators*, for succeeding BBs. Thus, a guard may be an "or" conjunction of preceding activators and evaluates to true as long as one incoming activator is also true.

Let S be the set of states of a program P. Let $B \subseteq 2^S$ be the set of all BBs. $P.head(b)$ is true if b is the first BB of P. Each $s \in S$ is contained in exactly one $b \in B$, i. e., it is

$$\forall b_1, b_2 \in B : b_1 = b_2 \text{ or } b_1 \cap b_2 = \varnothing$$

and

$$\forall s \in S : \exists b \in B \text{ s.t. } s \in b$$

For $b \in B$ define $b.last \in b$ as last statement of b. If $b.last$ is a conditional statement, $succ_{cond}(b, b') = \{b_{true}, b_{false}\}$ and defines whether b' is in the *true* or the *else branch* of b. $lcsexpr(b, b')$ returns the expression of $b.last$, denoted $exp(s_{last})$, if $succ_{cond}(b, b') = b_{true}$ and $\neg exp(s_{last})$, if $succ_{cond}(b, b') = b_{false}$. If $b.last$ is not a conditional instruction, $lcsexpr(b, b')$ returns true.

A BB $b_1 \in B$ is a *Basic Block Predecessor (BBP)* of another BB $b_2 \in B$, with $b_1 \neq b_2$, if any statement $s_1 \in b_1$ has an outgoing control flow edge to any statement $s_2 \in b_2$. Due to the enforced rules s_2 will be the first statement in b_2.

Fore some $b \in B$, let $guard(b)$ be the corresponding guard and $pred(b) \subseteq B$ be the set of BBs immediately preceding b.

$$gexp(b) := \{g' \wedge lcsexp(b', b) | b' \in pred(b), g' = guard(b')\}$$

If b starts with a depth statement and b' is the corresponding surface statement, $gexp(b) := pre(g')$ where $g' = guard(b')$.

Thus, the logical "or" combination of all guard expressions combined with the GO signal, if b is the first BB in the program, determines the activity state of a BB b, denoted $active(b)$.

$$active(b) := (\bigvee_i g_i \in gexp(b)) \vee (GO \wedge P.head(b))$$

Generally speaking, a BB is active, if and only if at least one of its guard expressions is active in the same tick instance. Additionally, a BB may pose as predecessor and hence, comprises a set of successor blocks. A succeeding BB is called *Basic Block Successor (BBS)*. For convenient readability all BBs are enumerated beginning at the start of a given SCL program and pause surface guards from previous ticks are prefixed with pre_.

The Join Synchronizer

The activation criterion for the join of concurrent threads is a little more complex since it is only permitted to proceed if all preceding threads are terminated and at least one of them exited in the actual tick instance. The construction of the synchronizer is done similar to the synchronizer circuit described in the "The Esterel v5 Language Primer, Version v5" [Ber00]. Here, each thread status is signaled by an *empty flag* which

describes whether or not a thread is not active. All empty flags are combined in a conjunction together with a combination of exit codes taht signal whether at least one thread terminated in this tick instance. The empty flag is combined with the go signal of the preceding circuit to detect active instantaneous threads.

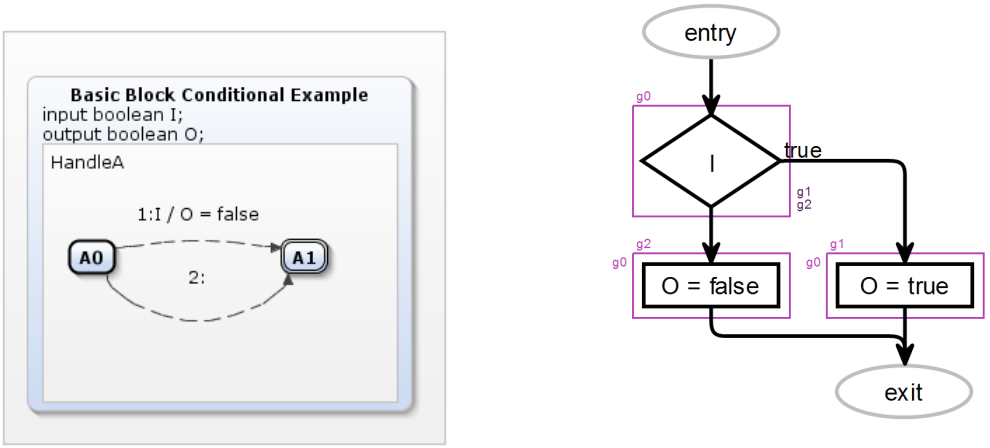
> Let $T' \subseteq T$ be the set of all concurrent threads of an common fork-join construct. For every thread $T_i \in T'$, $B_i \subseteq B$ holds all BBs of that thread. Let further $\beta_i \subseteq B_i$ be the set of BBs that indicate a state currently selected for resumption, i. e., an internal pause register. A thread is denoted as *empty*, if none of its blocks nor the predecessor of the fork, denoted by $pred(T_i.fork)$, is active in this tick instance. The empty state of T_i is defined as
>
> $$empty(T_i) \quad := \quad \neg(\bigvee_j \{active(b_j) | b_j \in \beta_i\}$$
>
> Let $\gamma_i \subseteq B_i$ be the set of BBs that reach the exit node $t_i.exit$ of T_i immediately. The activation criteria $active_{join}(b), b \in B$ for a BB b, whose first statement $s.head$ is a ParallelJoin, depends on the empty states of all preceding concurrent threads T_i and also holds the condition that at least one of the threads exited in the actual tick instance. Thus, $active_{join}(b)$ is defined as
>
> $$\begin{aligned} active_{join}(b) \quad := \quad & (\bigwedge_i \{empty(T_i) \vee (\bigvee_j \{active(b'_j) | b'_j \in \gamma_i\})\}) \wedge \\ & (\bigvee_i \{\bigvee_j \{active(b'_j) | b'_j \in \gamma_i\}\}) \end{aligned}$$

Basic Block Examples

The next two figures exemplify the visualization of BBs in an SCG. Figure 4.22b shows a BB split at a conditional instruction. It is the first instruction in the program and hence is marked as g0. Depending on the evaluation of the expression of the conditional the control forks off and O is set to true or false. Both blocks see g0 as a BBP and depict it as such on the left upper side. Analogously, g1 and g2 are possible successor blocks to g0 and are listed at the bottom right side of g0.

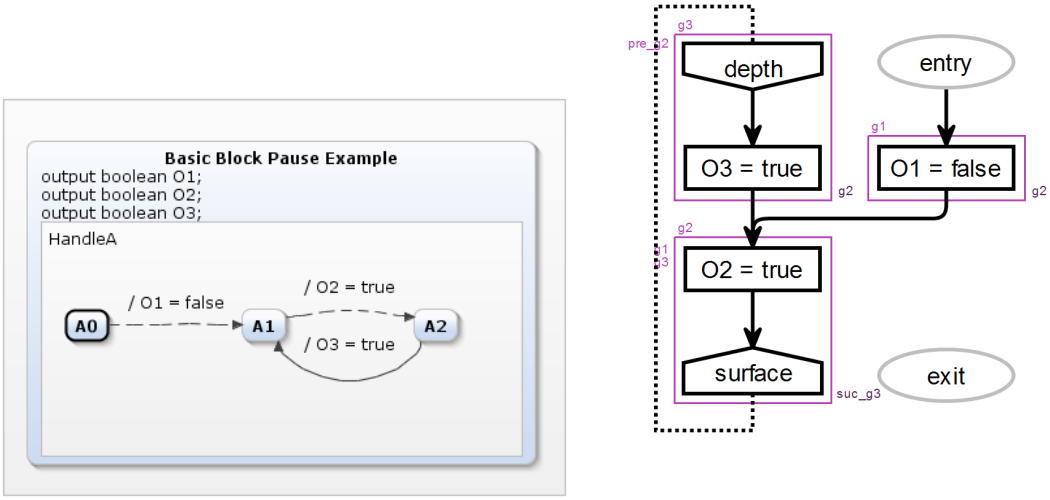
In the second example, Figure 4.23b, the first BB is broken off at the join of two control flows. After O1 is set to $false$ in g0 the control flow merges with the loop of g1 and g2. These two blocks are separated, because of the pause split mentioned in earlier in Section 4.2.6. Since g2 depends on the state of g1 in the tick before, the predecessor is prefixed with pre_ and the corresponding successor is prefixed with suc_. As g1 and g2 compose an infinite loop, the exit node is never reached.

Figure 4.24b shows a simple concurrent SCG. A parallel forks off two threads with an assignment to an output variable each. Both assignments contained in the BBs g2 and g3 depend on the block g0 guarding the fork node. Consequently, they are active immediately, when the fork node activates. As defined in the previous section, the join, guarded by g1, is only permitted to proceed if all threads are terminated and at least



(a) SCChart **(b)** SCG

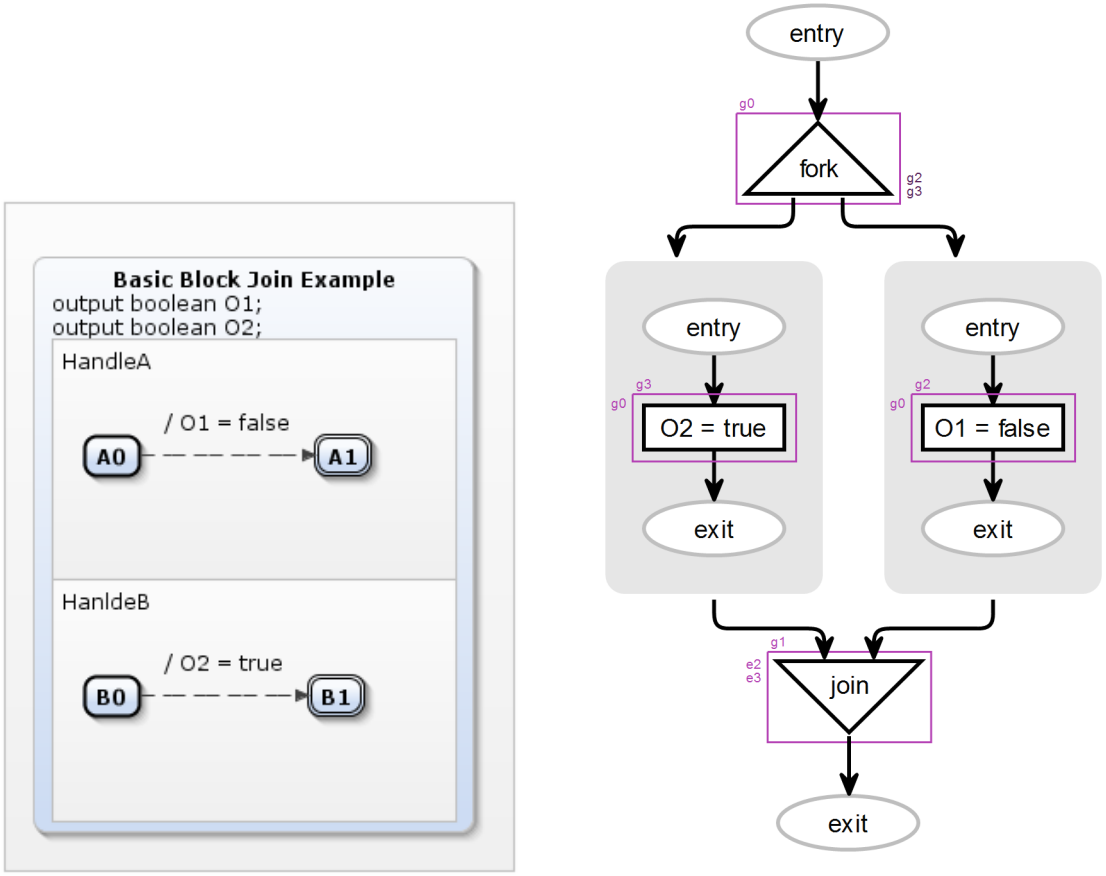
Figure 4.22. Basic Block Conditional Example



(a) SCChart **(b)** SCG

Figure 4.23. Basic Block Pause Example

one is exited in this tick. As this is a special case, the activators for this BB are depicted with an **e** instead of a **g**. In the context of concurrent joins, the first BB of a thread identifies the thread and is listed in the guard illustration on the upper left side. As earlier discussed a more complex synchronizer is created at a concurrent join. Hence, the guard predecessors are listed for convenient reasons only.



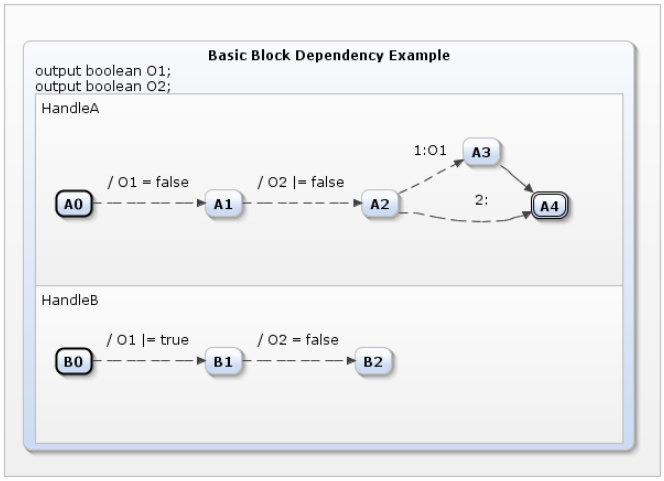
(a) SCChart **(b)** SCG

Figure 4.24. Basic Block Join Example

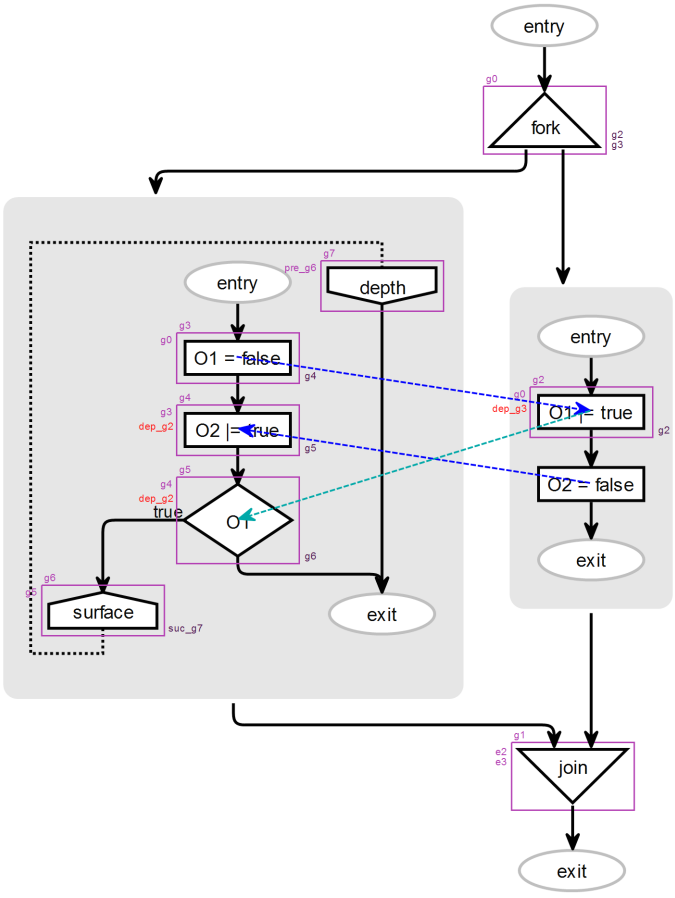
Basic Block Data Dependencies

Besides guards depending on each other a BB may hold concurrent data dependencies. If a variable access va_1 inside a BB b_1 with guard g_1 depends on an access va_2 in another BB b_2 with guard g_2, b_1 inherits the dependency from its variable access va_1. A *data dependency* does not alter the activity state of a BB. However, it imposes an ordering constraint and determines the sequence of guard evaluations since g_2 must be evaluated before g_1. If data dependencies are cyclic, the program is not *ASC-schedulable*.

Figure 4.25b exemplifies an SCL program with two concurrent threads t_l and t_r. They are created by the fork instruction and comprise interleaving data dependencies. Theoretically both threads may proceed immediately after the control flow reaches the fork node. However, following the rules imposed by the SC MoC, absolute writes are scheduled before relative ones. Thus, t_l must execute his first assignment O1 = false before t_r may proceed. In fact, the program is compelled to reschedule at this time since



(a) SCChart



(b) SCG

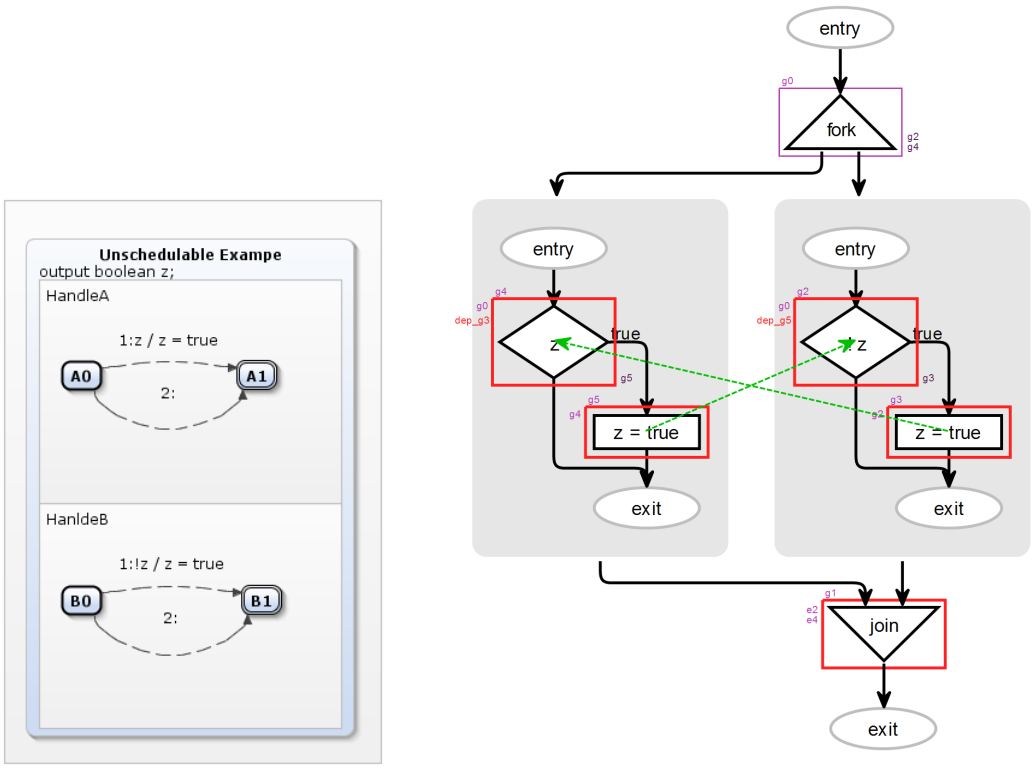
Figure 4.25. Basic Block Data Dependency Example

t_l must wait for t_r to execute its assignments. Only after t_r finishes the absolute write of O2, t_l may proceed with its second assignment. Both relative dependencies are illustrated by the blue directed edges in Figure 4.25b. Even though the data dependencies of the program do not modify the activation state of the guards and both threads may proceed simultaneously, a strict scheduling order between the two threads is imposed.

As shown in Figure 4.25b, data dependencies are depicted slightly brighter on the left side of a BB subsequent to the BBP list.

Unschedulable Basic Blocks

If dependencies are cyclic, the BB guards cannot be calculated. Hence, the program is not ASC-schedulable as defined in Section 4.1.2. To aid in the detection of unschedulable blocks, the SCG marks all BBs containing potential conflicts bright red.



(a) SCChart **(b)** SCG

Figure 4.26. Basic Block Conflict Example

Figure 4.26b depicts the visualization of potential conflict. Although guard g0 may activate due to the GO signal, the control flow cannot proceed, due to the interleaving dependencies. Hence, all subsequent guards contain conflicts, since they cannot be calculated.

4.2.7 Sequential SCL Transformation

If an SCL program is ASC-schedulable, a generic tick function can be generated. As discussed in Section 4.1.6 an SCL program in its sequential form is stripped of any concurrency and registers. Then, the tick function is executed in each tick instance and calculates the control flow of the corresponding SCL program depending on the actual state.

```
1  module abo_tick
2  boolean A, B, O1, O2, GO;
3  boolean g0, g3, e4, e8, g4, g5, g6;
4  boolean g7, g8, g9, g10, g11;
5  boolean g6_pre, g8_pre;
6  {
7     if GO then
8        g6_pre = false;
9        g8_pre = false;
10    end;
11    g0 = GO;
12    if g0 then
13       O1 = false;
14       O2 = false;
15    end;
16    g7 = g6_pre;
17    g9 = g8_pre;
18    g4 = g0 || g7;
19    g5 = g4 && A;
20    if g5 then
21       B = true;
22       O1 = true;
23    end;
24    g6 = g4 && ! A;
25    g10 = g9;
26    g11 = g10 && B;
27    if g11 then
28       O1 = true;
29    end;
30    g8 = g0 || ( g10 && ! B );
31    e4 = ! ( g6 );
32    e8 = ! ( g8 );
33    g3 = ( g5 || e4 ) && ( g11 || e8 )
34       && ( g5 || g11 );
35    if g3 then
36       O1 = false;
37       O2 = true;
38    end;
39    g6_pre = g6;
40    g8_pre = g8;
41 }
```

Listing 4.16. ABO in Sequential SCL

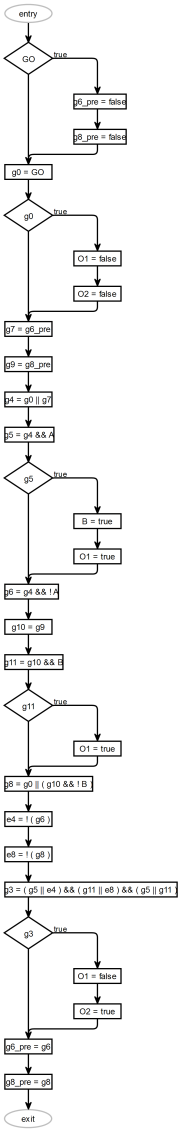


Figure 4.27. SCG of Seq SCL ABO

Therefore, every guard of all basic blocks of an SCL program is evaluated with respect to any imposed dependency constraint. If a basic block contains any assignments, the transformation subsequently creates a conditional statement with the guard as expression and the assignments as statement sequence. Lastly, all _pre guards are set.

For instance, the sequential SCL program for ABO is shown in Listing 4.16. Its corresponding SCG is depicted in Figure 4.27. As being a valid SCL program, all definitions are done in the first section before the main statement sequence. It comprises the original variables of ABO, since they represent the interface to the environment and also new boolean definitions for each guard and the GO signal.

In the statement sequence each guard is calculated with respect to any BB constraint detected in the preceding analyses. Meaning, a guard is allowed to be processed if and only if all of the guards it depends on are already calculated. An example of this rule is guard g4 in line 12. It depends on g0, which is calculated in line 5 and guard g7, line 10. Thus, g4 must be evaluated after line 10.

The conditionals encapsulate any original variable assignments and are guarded by the guard of their BB. Line 14 comprises the conditional which includes the emission of B and O1. It is guarded by g5 and g5 depends on g4 and A. This represents the transition outgoing from state WaitA in ABO, if A is present. To validate this, as depicted in the SCChart of ABO, B and O1 are in fact emitted once A is present in this state.

Lines 24 - 27 illustrate the generated synchronizer. The empty flags are denoted with an e as described in Section 4.2.6. They are only true if the corresponding guard of their threads are inactive. Finally, g3, the guard guarding the join, is comprised testing for any thread to be empty or just terminated.

4.2.8 SCL to S Transformation

Since the generic tick function is meant to be an initial point for further software or hardware synthesis, one possible way to generate binary code is the translation to SC. Furthermore, as this approach is embedded in the KIELER framework, the resulting executable code can be simulated directly in KIEM since it already comprises a simulator for SC. Moreover, the results of the simulation are comparable to semantically identical SyncCharts, due to the fact that they can also be translated to SC and simulated in KIEM.

Chapter 6, Experimental Results, discusses test setups for different simulations of the two statechart types in KIEM. It summarizes the results of these simulations and subsequently draws conclusions of this comparison.

To simulate the generic tick function in the KIELER environment, the sequential SCL is translated into S. Therefore all guards are transformed to valued S signals of the type boolean and two states are created. The first one, named _go represents the starting state, emits the GO signal and transits to the tick state _tickStart. This state comprises the actual generic tick function and loops in every macro step. Each guard assignment translates into an emit and every conditional corresponds to an if-construct.

The S program of ABO is depicted in Listing 4.17. As mentioned before, it contains

two states, the initialization state _go and the state for the tick function _tickStart. Since the signals are emitted in each tick instance, the activation state of the guard is carried in the value of that state. A value of a valued signal is queried by a question mark (?).

As shown in the listing, the translation of the guard calculation is done straightforwardly and results in relatively big emit constructs. The pause instruction in line 41 marks the end of the tick function before it is restarted again by the trans, line 42, due to the traversal to the _tickStart state.

```
1  synchronous program abo_tick ( 1 )
2
3  // signals skipped
4
5  state ( _go ) {
6     emit ( GO ( true ) );
7     trans ( _tickStart );
8  }
9
10 state ( _tickStart ) {
11    if ( ?GO = true ) {
12       emit (g6_pre(false));
13       emit (g8_pre(false));
14    };
15    emit ( g0(?GO) );
16    if ( ?g0 = true ) {
17       emit (O1(false));
18       emit (O2(false));
19    };
20    emit (g7(?g6_pre));
21    emit (g9(?g8_pre));
22    emit (g4(?g0 or ?g7));
23    emit (g5(?g4 and ?A));
24    if ( ?g5 = true ) {
25       emit (B(true));
26       emit (O1(true));
27    };
28    emit (g6(?g4 and not ?A));
29    emit (g10(?g9));
30    emit (g11(?g10 and ?B));
31    if ( ?g11 = true ) {
32       emit(O1(true));
33    };
34    emit (g8(?g0 or ?g10 and not ?B));
35    emit (e4(not(?g6)));
36    emit (e8(not(?g8)));
37    emit (g3(( ?g5 or ?e4 ) and ( ?g11 or ?e8 ) and ( ?g5 or ?g11 )));
38    if ( ?g3 = true ) {
39       emit ( O1(false) );
40       emit ( O2(true) );
41    };
42    emit (g6_pre(?g6));
43    emit (g8_pre(?g8));
44    pause ();
45    trans ( _tickStart );
46 }
```

Listing 4.17. ABO in S

Chapter 5

Sequential Constructiveness Code Generation Implementation

Chapter 4 explained the overall approach behind the SCL code generation presented in this thesis. This chapter will describe the mandatory implementations for this approach. It is structured in four parts. Section 5.1 explains how the SCL and its corresponding metamodel, both discussed in Section 4.1.3, were created. Additionally, the section will describe technical details about dynamic extensions of the metamodel discussed in Section 4.1.5. Subsequently, the third part of this chapter, Section 5.3, will illustrate how an SCG is synthesized out of an SCL model and how the information gathered by the SCL extensions is displayed graphically. Finally, the chapter closes with the core transformations discussed in Section 4.2.

5.1 The Sequentially Constructive Language

With SCCharts described in detail by Duderstadt [Dud12] in 2012 and S, mentioned in Section 3.2.4, still being in development by Motika at the time of writing, this thesis mainly covers the SCL and its graphical representation, the SCG. As already mentioned in Section 4.1.3, the language itself provides all mandatory infrastructure to handle and store model related data. The engineering of the SCL is described in Section 5.1.1.

Nevertheless, to aid in implementations a number of extensions which are discussed in Section 5.2 have been added to the SCL metamodel.

5.1.1 SCL Grammar in Xtext

As discussed in Section 4.1.3 SCL is a DSL particularly designed for the representation and analysis of SCCharts. Its abstract syntax is established by the SCL metamodel. The metamodel is defined by a grammar within the Xtext framework and subsequently Xtext automatically derives the SCL metamodel. As described in Section 3.1.4 the Xtext framework also generates a parser, a serializer and a corresponding full-featured editor once a specialized grammar is defined. This section exemplifies the different grammar rules of the SCL definition.

Since the approach utilizes the expressions and types, elucidated in Section 4.1.3, included in the KIELER extension of Yakindu, introduced in Section 3.3, the SCL grammar is derived from the KIELER Yakindu implementation *SyncText*. The SCL grammar

definition is located in the SCL.xtext resource stored de.cau.cs.kieler.scl plugin and starts with the rule shown in Listing 5.1.

```
1  grammar de.cau.cs.kieler.scl.SCL
2      with de.cau.cs.kieler.yakindu.sccharts.model.stext.Synctext
3
4  // imports [...]
5
6  Program:
7      'module' name = ID
8      (definitions+=VariableDefinition)*
9      '{'
10     (
11         ((statements += InstructionStatement';') | statements += EmptyStatement)*
12         (statements +=  InstructionStatement statements += EmptyStatement*)?
13     )
14     '}'
15 ;
```

Listing 5.1. SCL Grammar – Program root

Section 4.1.3 discusses the metamodel elements needed to model a valid SCL program. Since each SCL program starts with a program root element, the first rule in the grammar is the program rule and marks the entry point of the model. As depicted in lines 5 to 14 in Listing 5.1 a program starts with the keyword module followed by the name of the program. Optionally, a list of global variable definitions may be added. Subsequently, the source code of the program is encapsulated in curly brackets and is stored in the statements member. Since instructions are linked with the sequence operator, while labels are not, statements that comprise an instruction are separated by a semicolon. To model the sequence operator correctly the last instruction is not obliged to finish with a semicolon. However, for reasons of convenience sequence operators followed by no further instruction are implicitly filled with a *no operation* instruction and allow the modeler to close programs with semicolons.

The VariableDefinition is derived from *SyncText*, depicted in Listing 5.2. A variable may be prefixed with *input*, *output* or *static*. As usual in the real-time context, input variables get information from and output variables feed data back to the environment. Local variables marked as *static* are only initialized once, when their scope is entered and persist in consecutive calls. Furthermore, every variable has a type. It is derived from the

```
46 VariableDefinition:
47     (input?='input')? (output?='output')? (static?='static')?
48     type=[types::Type|FQN] name=ID
49     ('=' initialValue=Expression)?';'
50 ;
```

Listing 5.2. SyncText Grammar – Variable Definition

Yakindu framework and even though this thesis focuses around boolean and integer types, Yakindu allows other primitive types such as floats and strings and provides extension points for custom types. Eventually, the name of the variable must be specified with an optional initial value.

The SCL definition in Section 4.1.3 states that programs and nested hierarchies are comprised out of statement lists. Statements are either an InstructionStatement containing a specific SCL instruction or an EmptyStatement comprising a label. The two types have been separated mainly because an arbitrary number of labels may follow consecutively without any instruction, induced through several final states included in a thread. These are translated to labels only as described in Section 4.2.2. Both statement types together form the statement rule and may contain an arbitrary number of annotations, which are explained in the next section.

Listing 5.3 shows the rules that define statements in the SCL metamodel. Lines 21 to 24 depict the EmptyStatement including its annotations and the label, whereas the InstructionStatement is defined in lines 26 to 29. The statement combination of both is shown at the beginning of the listing.

```
17 Statement:
18     EmptyStatement | InstructionStatement
19 ;
20
21 EmptyStatement:
22     (annotations += Annotation)*
23     (label = ID ':')
24 ;
25
26 InstructionStatement:
27     (annotations += Annotation)*
28     instruction = (Assignment | Conditional | Goto | Parallel |
29                    Pause | StatementScope)
30 ;
```

Listing 5.3. SCL Grammar – Statements

Section 4.1.3 also introduces all SCL necessary to represent SCCharts in textual form. Since it is desired to model SCL with exactly these instructions, they must be defined in the grammar. Each SCL instruction defined in Section 4.1.3 is represented by one grammar rule in the SCL grammar. Listing 5.4 shows the first part of the instruction rules.

Since the SCL program treats all SCL instructions as some kind of instruction, all instruction rules together comprise the instruction rule. The amalgamation of instructions is shown in lines 31 to 33. It refers to the particular instruction rules and separates the instruction with disjunctions.

Assignment: To model an assignment defined as $x = e$ with e being any expression, SCL makes use of the expression language provided by the Yakindu framework. Since the

```
31 Instruction:
32    Assignment | Conditional | Goto | Parallel | Pause | StatementScope
33 ;
34
35 Assignment:
36    assignment = Expression
37 ;
38
39 Goto:
40    'goto' targetLabel = ID
41 ;
42
43 Pause:
44    'pause' {Pause}
45 ;
46
47 StatementScope:
48    {StatementScope}
49    '{'
50    (definitions+=VariableDefinition)*
51    (
52       ((statements += InstructionStatement';') | statements += EmptyStatement)*
53       (statements +=  InstructionStatement statements += EmptyStatement*)?
54     )
55    '}'
56 ;
```

Listing 5.4. SCL Grammar - Instructions

grammar is derived from SText, Expression in the assignment rule, lines 35 to 37, is immediately available.

Goto: An SCL goto instruction is defined as a jump to a specific label identified by an ID. Since the ID shall be editable freely in the text editor, the restricted character set of the Xtext framework ID can be used instead of an object reference.

In the grammar the goto rule, lines 39 - 41, begins with the goto keyword and is followed by the ID which identifies the target label.

Pause: The SCL pause instruction simply marks a tick boundary and it does not hold any further information. It only comprises the pause keyword in the grammar.

Statement Scope: A statement scope introduces a new scope layer to allow the definition of local variables and to structure the SCL code. The syntax of variables definitions differs slightly from the syntax of global variables as it is encapsulated within the curly brackets to indicate the scope. Therefore, these two kinds of variable definitions must be separated in different rules. Besides the difference in the syntax for reasons of readability, the definition of variables is analogous to the one defined in the program rule.

The second part of the grammar comprising the definitions for the SCL grammar is

depicted in Listing 5.5. It shows the rules for the conditional and parallel instructions and threads as discussed in Section 4.1.3.

Conditional: SCL conditionals are defined as if e then s_1 else s_2 end. Therefore, they must be able to test an expression and comprise two distinct statement lists, one for the true case and the other for the else branch.

In the grammar, a conditional is introduced by the if keyword and followed by an SText expression similar to the assignment rule. The control flow proceeds in the "then branch" if the expression evaluates to true. Otherwise, the "else branch" is taken if present. In the grammar the statement list is built analogously to the program statement list. The "else branch" is optional and denoted elseStatements.

Thread: Since SCL allows an arbitrary number of concurrent threads, an object which holds a single thread is necessary. Hence, a thread comprises a list of statements without any further syntactical elements. This permits other rules, in particular the succeeding parallel rule, to include them without restriction.

In the grammar it is defined like any other statement list.

```
58 Conditional:
59    'if' expression = Expression 'then'
60    (
61       ((statements += InstructionStatement';') | statements += EmptyStatement)*
62       (statements +=  InstructionStatement statements += EmptyStatement*)?
63    )
64    ('else'
65      (
66        ((elseStatements += InstructionStatement';') |
67           elseStatements += EmptyStatement)*
68        (elseStatements +=  InstructionStatement elseStatements += EmptyStatement*)?
69      )
70    )?
71    'end'
72 ;
73
74 Thread:
75    {Thread}
76    (
77       ((statements += InstructionStatement';') | statements += EmptyStatement)*
78       (statements +=  InstructionStatement statements += EmptyStatement*)?
79    )
80 ;
81
82 Parallel:
83    'fork'
84    (     threads += Thread
85    ('par'
86          threads += Thread)*)
87    'join'
88 ;
```

Listing 5.5. SCL Grammar – Instructions (cont.)

Parallel: Parallel instructions fork off the control flow and allow an arbitrary number of threads to proceed. To model this the preceding thread rule is utilized.

A parallel instruction starts with the fork keyword and is followed by at least one thread. Multiple threads are separated by par and are stored in the thread member of the object. Eventually, the instruction is closed by the join keyword.

Finally, to consolidate all statement lists in one object a rule for a Statement Sequence is created. Even though their generated syntax may differ slightly, all objects including a statement sequence are derived from the statement sequence class and hence, can be treated the same in the context of statement lists.

```
90 StatementSequence:
91    Thread | Program | Conditional | StatementScope
92 ;
```

Listing 5.6. SCL Grammar – Statement Sequence

Annotation Grammar and Highlighting

Statements are allowed to carry an arbitrary number of annotations. They are described in Section 4.1.3 and an example of an annotation with syntax highlighting is depicted in Figure 4.9.

In the grammar, annotations are defined by the Annotation rule. An annotation is introduced by an @ symbol and followed by a mandatory keyword and an optional list of parameters. The annotation mechanism used in this implementation is very light-weight and at time of writing only used to carry specific layout information for the SCG visualization that deviates from the default settings.

To add highlighting for additional elements a highlight configuration and calculation class is added to the User Interface (UI) plug-in. The two classes must be bound in the initialization of the UI plug-in. The calculation class searches for elements to highlight and consults the configuration for information how to highlight a specific element.

SCL uses the custom highlighting to point out annotations. The calculation uses the automatically generated SCLSwitch to search for model elements. If an annotation is found, it is formatted with the configured *TextStyle*. Annotations in the SCL are printed bold, italic and in a dark green.

```
91 Annotation:
92    '@' name = ID
93    (':' parameter += ID(',' parameter += ID)*)?
94 ;
```

Listing 5.7. SCL Grammar – Annotation

Technical Details

Once all rules are defined Xtext may be invoked to create the SCL metamodel, the parser and serializer elements and an SCL text editor with default settings for highlighting and auto-completion. While the language elements are stored in de.cau.cs.kieler.scl all UI components are saved to de.cau.cs.kieler.scl.ui.

To enable *backtracking*, the Xtext generation configuration file has to be edited. Here, find the fragment entry of the parser generator and set the option backtrack to true as depicted in Listing 5.8.

```
fragment = parser.antlr.XtextAntlrGeneratorFragment {
    options = {
        backtrack = true
    }
}
```

Listing 5.8. Xtext configuration – Parser Fragment

Projects using the SCL can easily make use of all the generated tools. For instance, the serializer in the the SCG implementation is requested by the two lines depicted in Listing 5.9.

```
    private static val Injector i = SCLStandaloneSetup::doSetup();
    private static val ISerializer serializer = i.getInstance(typeof(ISerializer));
```

Listing 5.9. SCL – Request serializer

5.2 Dynamic Language Extensions

The SCL metamodel defined in Section 4.1.3 and construction via Xtext in Section 5.1 represents the abstract syntax of the SCL. However, to work more efficiently with the infrastructure provided through the metamodel, Section 4.1.5 defines a number of extensions. These can be implemented via Xtend as described in Section 3.1.5.

This section describes the two most important extensions of the SCL since they are mandatory to execute the dependency analysis, Section 4.2.5, and the Basic Block analysis, Section 4.2.6. The first part elucidates the SCL dependency extension. It describes the methods required to find variable accesses in concurrent threads and categorizes the accesses with respect to their conflict potential as discussed in Section 4.1.2. Threads are considered concurrent, if and only if they share a common ancestor fork, defined in Section 4.2.5.

Secondly, the section focuses on the SCL Basic Block extension. It explains how BBs are composed according to the rules imposed by Section 4.2.6. Additionally, methods

for finding BBPs are presented. Subsequently, the section provides information about optimization of the BB extension with respect to efficiency.

The interested reader will find the implementation of the other extensions in the KIELER semantics repository in the de.cau.cs.kieler.scl.extensions package.

5.2.1 The SCL Dependency Extension

As mentioned in the introduction of this section, the Dependency Extension is responsible for finding and categorizing dependencies in an SCL model according to the rules imposed by Section 4.1.2. To fulfil this task all expressions included in the model of the program have to be examined. Since the SCL metamodel provides all mandatory information, the examination is executed directly on the model in question.

This section depicts the methods necessary to accomplish the dependency analysis. Firstly, it describes how *expression references* to variable definitions of a model, introduced in Section 4.1.3, are found. Secondly, the query of all statements which comprise these references is shown. As mentioned in Section 4.2.5 accesses are only potentially conflicting if they occur in a concurrent context. The determination of concurrent references is explained in the third part. Finally, all found conflicting variable accesses are categorized according to their severity consistent to the sequential rules imposed in Section 4.1.2.

Dependency References

Firstly, the getDependencyReferences method, depicted in Listing 5.10, returns all REFEXP, introduced in Section 4.1.3, found in an expression. As defined in the SCL metamodel, expressions occur in assignments and conditionals. The method is used to retrieve all references to variables used in their expressions.

At the beginning of the function an empty list of REFEXP is instantiated. Then, if the expression itself is already an REFEXP, it is added to the list, since eAllContents only retrieves comprising elements and not the element itself. Subsequently, all containing objects are filtered and all REFEXP are appended to the list. Finally, the list is returned. The method is overloaded to retrieve the references of an instruction directly.

```
1 def List<ElementReferenceExpression> getDependencyReferences(Expression expr) {
2     val erexList = new ArrayList<ElementReferenceExpression>
3     if (expr instanceof ElementReferenceExpression) {
4         erexList.add((expr as ElementReferenceExpression))
5     }
6     expr.eAllContents.toIterable.filter(typeof(ElementReferenceExpression)).
7         forEach(e | erexList.add(e))
8     erexList
9 }
```

Listing 5.10. SCL Dependency Extension – References Search

```
11 def List<Statement> getDependencyStatements(StatementSequence sequence,
12    ElementReferenceExpression referenceExpression) {
13    val statementList = createNewStatementList()
14    for(instruction : sequence.eAllContents.filter(typeof(Instruction)).toList) {
15       if ((instruction instanceof Assignment) ||
16          (instruction instanceof Conditional)) {
17          val references = instruction.dependencyReferences
18          for (reference : references) {
19             if (reference.reference.equals(referenceExpression.reference))
20                if (!statementList.contains(instruction)) {
21                   statementList.add((instruction.eContainer as Statement));
22             }
23          }
24       }
25    }
26    statementList
27 }
```

Listing 5.11. SCL Dependency Extension – Statements Search

Dependency Statements Retrieval

Secondly, as stated in the introduction of this section, any statement that contains the queried references must be found. Therefore, the method getDependencyStatements, listed in Listing 5.11, returns a list of all statements which have a dependency to a given REFEXP. It searches for assignments and conditionals in a given statement sequence and fetches all element references in theses instructions. If an REFEXP is equal to the given REFEXP provided by the caller and if the reference is not already included in the return list, it is added to the list of statements. Finally, the statement list is returned.

Concurrent Dependency Query

With getDependencyStatements defined, a list of all statements which include references to a variable definition can be returned. As stated in Section 4.2.5 only concurrent conflicting variable accesses are problematic. Therefore, the Dependency Extensions need to find all conflicting statements in a concurrent context. This can be done by utilizing a *least common ancestor search* as also defined in Section 4.2.5.

Consequently, the getConcurrentDependencies method retrieves a list of statements which comprise a dependency to a corresponding concurrent statement. Listing 5.12 shows its structure. If the statement is an empty statement, no dependency exists and the empty list is returned. Otherwise, a list of dependencies containing all dependencies for the given statement in the affiliated SCL program is created. For every dependency the method checks whether or not the two instructions share a common ancestor fork node and are not in the same thread. If this requirement is true, a concurrent dependency is found and added to the statement list which is returned at the end of the function.

```
29 def List<Statement> getConcurrentDependencies(Statement statement) {
30    val statementList = new ArrayList<Statement>
31    if (statement.isEmptyStatement) {
32      return statementList;
33    }
34    val dependencyList = statement.dependencyInstructions(statement.getProgram)
35    for (targetStatement : dependencyList) {
36       if (!statement.isInSameThreadAs(targetStatement) &&
37          instruction.getLeastCommonAncestor(targetStatement.getInstruction) != null
38       ) {
39          statementList.add(targetStatement)
40       }
41    }
42    statementList
43 }
```

Listing 5.12. SCL Dependency Extension – Concurrent Dependencies Search

Dependencies Categorization

As discussed in Section 4.1.2 variable accesses are distinguished as reads and writes. Furthermore, a write can be an absolute or relative write. To categorize any found dependencies a set of helper methods examine the dependent expressions nested in the instructions.

As an example the function for the relative writer determination isRelativeWriter is illustrated in Listing 5.13. If a given instruction is not an assignment, the instruction is not a writer. Otherwise, the operator of the assignment is checked. If it is an *assign operator*, the reference of the variable that is written to must be found on the right hand side of the equation. Conservatively, the method evaluates to true if the expression on the right hand side is of relative nature. Since the implementation in this thesis focuses on boolean and integer types, checks for the logical expressions in the boolean case and for addition and multiplication in the integer case are sufficient. If the operator is already a relative assignment, the reference of the variable on the right side of the equation is optional.

```
47 def boolean isRelativeWriter(Instruction instruction) {
48    if (!(instruction instanceof Assignment)) return false
49    val assignment = (instruction as Assignment).assignment as AssignmentExpression
50    val reference = assignment.varRef
51    if (assignment.operator != AssignmentOperator::ASSIGN) return true;
52    if (assignment.expression.eAllContents.toIterable.filter(
53       typeof(ElementReferenceExpression)).filter(e|e.equals(reference)).size>0) {
54       if (assignment.isConfluent) return true
55    }
56    return false
57 }
```

Listing 5.13. SCL Dependency Extension – Relative Writer Determination

As stated in Section 4.1.2 a write access is absolute, if it is not relative. This is determined by the isAbsoluteWriter method. In addition, the function isConfluentWriter tests if two writers are confluent to one another. Two write accesses are confluent if both write to the same variable and assign the same value.

Dependency Type Determination

Finally, if a concurrent dependency is found, its type must be categorized according to the different types introduced in Section 4.1.2. Therefore, with all variable access determination methods present to identify concurrent accesses, the getDependencyType function determines the kind of a given access. The method tests relationships in the direction of a dependency edge. The first instruction is the source of the dependency and the second its target. Therefore, if it is called in the opposite direction of a dependency edge, the type will not be recognized and the dependency must be checked again with the correct instruction order.

If the *source* and the *target instructions* are absolute writers and both instructions are not confluent to each other, the dependency is a *write-write conflict*. Otherwise, if the source is an absolute write access and the target is a relative write access, while both are setting the same variable, the dependency is denoted as *write-increment*. If it is not a write-write relationship, but the second instruction reads the variable, a *write-read* dependency is found. If the source was a relative writer, the write-read dependency is called *increment read*.

The type *unknown* usually hints at an error in the extension since a dependency was found in the preceding analysis, but its type is not recognized. This may be the case if the method is called in the opposite dependency direction or if an expression was examined which is not yet supported. The first case is resolved by calling the method again with reversed parameters.

```
60 def getDependencyType(Instruction sourceInstr, Instruction targetInstr) {
61     if (sourceInstr.isAbsoluteWriter && targetInstr.isAbsoluteWriter &&
62         sourceInstr.getWriteReference == targetInstr.getWriteReference &&
63         !isConfluentAbsoluteWriter(sourceInstr, targetInstr))
64             return DependencyType::WRITEWRITE
65     if (sourceInstr.isAbsoluteWriter && targetInstr.isRelativeWriter &&
66         sourceInstr.getWriteReference == targetInstr.getWriteReference)
67             return DependencyType::WRITEINCREMENT
68     if (sourceInstr.isAbsoluteWriter &&
69         targetInstr.isReader(sourceInstr.getWriteReference))
70             return DependencyType::WRITEREAD
71     if (sourceInstr.isRelativeWriter &&
72         targetInstr.isReader(sourceInstr.getWriteReference))
73             return DependencyType::READINCREMENT
74     return DependencyType::UNKNOWN
75 }
```

Listing 5.14. SCL Dependency Extension – Dependencies Categorization

5.2.2 The SCL Basic Block extension

The second mandatory schedulability analysis is the BB analysis explained in Section 4.2.6. It structures SCL statements in related blocks with respect to their control flow and possible context switches.

This section elucidates the creation of these blocks and how to find predecessors in the control flow, since they are necessary for the guard calculation defined in Section 4.2.6. Additionally, the Basic Block class is introduced, which aids in the handling of BB constructs and improves the efficiency of the analysis.

Basic Block Assemblage

The main method for finding BBs is getBasicBlockStatements. It searches all statements which belong to a BB and returns them in a list. As defined in Section 4.2.6 a statement can only be present in a single BB exclusively. Thus, the block in question is identified by any one of its statements. Furthermore, to find all BBs in an SCL program, the function getAllBasicBlocks just needs to iterate through all statements and to call getBasicBlock-Statements for each. The resulting list of BBs includes exactly all BBs of the program, since duplicated blocks are ignored.

Lets take a closer look at Listing 5.15 which illustrates getBasicBlockStatements in pseudo code. It expects any statement of the desired BB. Any mandatory instruction split, as described in Section 4.2.6, preceding the invocation of getBasicBlockStatements may indicate the position of the control flow with the isDepth flag. If the statement is a goto, it is seen as related to its predecessor. GetPreviousStatementHierarchical retrieves the predecessor of a statement respecting hierarchies. Subsequently, getBasicBlockStatements is called recursively with the previous statement. Otherwise, a new list of statements basicBlock for the BB is instantiated.

The list that comprises the statement in question is stored to statementList and the *index* of the statement in that list is saved in sIndex. Then, if the statement is not already the head of the BB all preceding statements must be added to the statement list until the head is found. Also, if the statement is a surface of a pause or a fork of a parallel, there may be additional preceding statements. The counter variable prevIndex runs from sIndex down to zero and searches for the head of the BB. It is sufficient to check the actual statement sequence only since hierarchical inferior sequences always introduce a new BB. Each statement is added to the list of statements until the actual head is found. Afterwards the provided statement is also added to the BB. If this statement is already the tail of a BB, the method is finished at this point unless it is executed in a depth context. In this case it may include statements of the surface of the succeeding control flow which are then added to the block. This case is flagged by breakAtHead.

Analogously to the preceding statements succIndex counts the succeeding statements, denoted succStatement in each iteration. If it is eligible to pose as new BB but is not a pause or a parallel while iterating in a depth indicated by breakAtHead, the loop is broken

```
def List<Statement> getBasicBlockStatements(Statement statement, boolean isDepth) {
   if statement.isGoto do
      // Create basic block list for corresponding statement
      previousStatement = statement.getPreviousStatementHierarchical
     return getBasicVlockStatements(previousStatement, isDepth)
   od
   basicBlock = createNewStatementList
   statementList = statement.getParentStatementSequence.statements
   sIndex = statementSequence.statements.indexOf(statement)
   // Add previous statements also to the block list
   if not statement.isBasicBlockFirst ||
      (statement.isPause && not isDepth && sIndex > 0) ||
     (statement.isParallel && not isDepth && sIndex > 0) do
     prevIndex = sIndex - 1
     while prevIndex >= 0 do
        prevStatement = statementList.get(prevIndex)
        if not prevStatement.isEmpty && not prevStatement.isGoto do
          basicBlock.insert(0, prevStatement)
         if not prevStatement.isBasicBlockHead do
          break
       od else do
          prevIndex = prevIndex -1
       od
     od
   od
   // Add the statement to the basic block
   basicBlock.add(statement)
   breakAtHead = false
   if statement.isBasicBlockTail do
      if not isDepth do return basicBlock
     // Surfaces of instruction splits may be included in this block
     breakAtHead = true
   od
   // Add succeeding statements to the list of this block
   succIndex = sIndex + 1
   while succIndex < statementList.size do
      succStatement = statementList.get(succIndex)
     if breakAtHead && succStatement.isBasicBlockHead &&
        not succStatement.isPause && not succStatement.isParallel do
       break
     od else do
        if not succStatement.isEmpty && not succStatement do
          basicBlock.add(succStatement)
       if not succStatement.isBasicBlockTail do
          succIndex = succIndex + 1
       od else do
          break
       od
     od
   od
   return basicBlock
}
```

Listing 5.15. SCL Basic Block Extension – Basic Block Statements Retrieval

and the method returns. Otherwise, as long as no tail is found, add the statements to the list of the BB and return subsequently.

Basic Block Decider

As depicted in getBasicBlockStatements, Listing 5.15, it is mandatory to decide if a statement may serve as initial or final point of a BB. The two elementary functions for detecting these crucial points are isBasicBlockHead and isBasicBlockTail. The first one decides whether or not a statement is the *head* of a BB whereas the second determines the possibility of a statement to serve as a *tail*.

IsBasicBlockHead, shown in Listing 5.16, is structured straightforwardly according to the rules imposed by Section 4.2.6. If the statement is empty or a goto instruction, it can not be the start of a BB. The statement sequence comprising the statement in question is queried and stored in statementSequence. Additionally, the previous statement respecting hierarchies is saved to prevStatement. Then, each following line decides whether the statement may pose as BB head.

If the statement is a parallel, a pause itself or the first statement in the sequence, the statement is the head of a basic block according to Section 4.2.6. Furthermore, if its predecessor is a conditional or a goto, the statement is also the head of a BB since the control flow forks off before the statement. Moreover, if the statement is an assignment or conditional that comprises additional incoming data dependencies or if goto jumps target the statement, it also initiates a BB since it has more than one incoming control flow or data dependency edge. As already mentioned an analogous method isBasicBlockTail exists to determine the tail of a BB.

```
1  def boolean isBasicBlockHead(Statement statement) {
2      if (statement.isEmpty) return false
3      if (statement.isGoto) return false
4
5      val statementSequence = statement.getParentStatementSequence
6      val prevStatement = statement.previousStatementHierarchical
7      if (statement.isParallel) return true
8      if (statement.isPause) return true
9      if (statementSequence.statements.indexOf(statement) == 0) return true
10     if (prevStatement.isConditional) return true
11     if (prevStatement.isGoto) return true
12     if ((statement.isAssignment || statement.isConditional) &&
13        statement.getInstruction.hasConcurrentTargetDependencies) return true;
14     if (statement.getIncomingGotos.size>0) return true
15     return false
16 }
```

Listing 5.16. SCL Basic Block Extension – Basic Block Head Statement decider

Performance Enhancement with the BasicBlock class

To simplify the handling of statement lists, instruction splits and caching the contents of BBs, the class BasicBlock composes all mentioned functionalities. It comprises a list of statements which represents the BB. Since the split of a pause results in two new objects PauseSurface and PauseDepth which contain references to the original pause instruction, they are included in the statement list without restriction. The extensions dereference the objects automatically when required. The same extension applies to the parallel statement.

As massive transient transformations consume comparatively more calculation time, the BasicBlock class encapsulates members for caching. Since they would transiently be recalculated at every call, the name of the block, its index and predecessors, successors and data dependencies are stored and must not be re-evaluated once they are cached.

The BasicBlock class also comprises member methods to compare blocks and to manipulate the statement list.

Basic Block Predecessors Determination

The guard calculation of BBs, defined in Section 4.2.6 and utilized by the Sequential SCL transformation in Section 4.2.7, needs information about the BBPs of each BB. To find the predecessors of a BB the method getPredecessors, shown in Listing 5.17 in pseudo code, returns a list of all BB which pose as predecessors of the BB in question.

At the beginning the list of predecessors is instantiated. If the block is a join of a parallel construct, all starting blocks of the corresponding threads pose as marker for the predecessors. Obviously, they are not the real preceding BBs and serve rather as visual information in the BB visualization since the transformation creates a more complex synchronizer. As no further blocks can be predecessors of the join, the method returns.

Secondly, if basicBlock starts with the depth of a pause, its only preceding block is the one containing the corresponding surface.

Otherwise, if it is neither a join nor a depth, the block may have multiple predecessors. To check for these, prevStatement holds the previous statement of the head of the block respecting hierarchy and prevImmediateStatement points to the immediate statement in the statement sequence regardless of any hierarchy. Hence, the head of the block being the first statement in a conditional branch will not have an immediate predecessor since the branch initiates a new statement sequence. In this case prevStatement will be a conditional and prevImmediateStatement will be empty. Furthermore, the actual BB is in fact the first block in a true branch of a conditional instruction. Thus, the preceding block comprising the parent conditional serves as predecessor.

If it is not a conditional, the preceding block may either be another BB in the same statement sequence, providing prevStatement is not empty or the fork node of a concurrent context. The fork node is eligible to pose as predecessor when the BB in question is the first block in any thread the fork node initiates. This is indicated by prevStatement

```
def List<BasicBlock> getPredecessors(BasicBlock basicBlock) {
   predecessors = createNewBasicBlockList
   if basicBlock.statements.head.isParallelJoin do
      // Add parallel threads as predecessors
      for each thread in basicBlock.statements.head.asParallel.threads do
        predecessors.add(thread.statements.head.getBasicBlock)
     od
     return predecessors
   od
   if basicBlock.getHead.isPauseDepth do
      // Add surface of the pause as predecessor
      predecessors.add(basicBlock.getHead.getBasicBlockBySurface)
     return predecessors
   od
   prevStatement = basicBlock.getHead.getPreviousStatementHierarchical
   prevImmediateStatement = basicBlock.getHead.getPreviousStatement
   if prevStatement.isConditional && prevImmediateStatement.empty do
      // Conditional
      predecessors.add(prevStatement.getBasicBlock)
   od else do
      if not prevStatement.empty && not prevStatement.isGoto do
        // Preceding basic block
        predecessors.add(prevStatement.getBasicBlock)
     od else do
        if prevStatement.empty && basicBlock.getHead.eContainer instanceof Thread
       do
          // Fork block
          predecessors.add(
               basicBlock.getHead.getForkEContainer.getBasicBlock)
       od
     od
   od
   if prevImmediateStatement.isConditional do
      // immediate Conditional
      trueBranchBlock =
        prevImmediateStatement.asConditional.statements.last.getBasicBlockByBranch
     if not trueBranchBlock.empty do
        predecessors.add(trueBranchBlock)
      elseBranchBlock =
        prevImmediateStatement.asConditional.elseStatements.last.
       getBasicBlockByBranch
     if not elseBranchBlock.empty do
        predecessors.add(elseBranchBlock)
   od
   if not basicBlock.getHead.isPauseDepth do
      for each goto in basicBlock.getHead.getIncomingGotos do
        predecessor.add(goto.getBasicBlock)
     od
   od
   return predecessors
}
```

Listing 5.17. SCL Basic Block Extension – getPredecessors in Pseudo Code

being empty and the head of the BB being contained in an SCL thread. If one of the preconditions is met, the required statement is queried and the corresponding BB is added to the list of predecessors.

Alternatively, if prevImmediateStatement is the conditional, the BB persists in the statement sequence which also includes the conditional. Thus, the control flow of the branches of the conditional may merge at this point. The last block of the true branch of the conditional is stored if trueBranchBlock and the last block of the else branch in elseBranchBlock. If a control flow of a branch does not merge due to including goto jumps, the corresponding block will be empty. Each not emtpy closing BB is added to the predecessors list.

Subsequently, if the block is not a pause depth, all goto jumps targeting the head of the block are added to the list of predecessors since they represent a merge of control flows. The method then exits and returns the list.

5.3 Synthesis of the Sequentially Constructive Graph

This section explains the basis of the SCG visualization explained in Section 4.1.4. It describes the traversal through statement sequences in the first part and depicts how corresponding figures are created. The section closes with the explanation of the implementation of the BB post-processing subsequent to the layouting of the SCG control flow.

The visualization of the SCG is triggered by KIVi, introduced in Section 3.2, when the SCL model is updated. As stated in Section 4.2.4 the SCG is generated transiently and not stored persistently. Every time the synthesis method is invoked, it creates figures for each instruction, draws dependency edges and calls a *Basic Block modifier* to insert BB information subsequently. Since it is not possible to discuss the source code of the synthesis in its completeness, this section focuses on the main loop, one exemplary figure function and the BB post-processing.

5.3.1 Statement Sequence Figures Creation

To visualize the SCG as discussed in Section 4.1.4 the synthesis traverses recursively through the SCL model defined by its metamodel, Section 4.1.3. Since all statements of an SCL program and including hierarchies are contained in a statement sequence, the main loop of the figures creation must process these lists. It examines the instructions in the list and decides which figures have to be created. Subsequently, the figures are connected with edges to represent the control flow of the SCL program.

The main loop of the SCG synthesis is contained in the createSequenceFigures method, depicted in Listing 5.18 in pseudo code. As indicated, an SCL program comprises a statement sequence, therefore, the method for creating the corresponding figures is called first once the synthesis has been invoked. However, since other instructions may comprise statement sequences themselves, it may be called several times recursively.

The method expects the statement sequence in question, a *root node* which represents the top level of the graph and if present an *entry node*, an *exit node* and an *outgoing port* for hierarchy and control flow forks. If an entry node is present, it functions as a

```
def createSequenceFigures(StatementSequence sequence, KNode rootNode,
   KNode entryNode, KNode exitNode, KPort outgoingPort) {
   precedingInstructions.Node = entryNode
   precedingInstructions.outgoingPort = outgoingPort
   for each statement in sequence.filter(InstructionStatement) do
   switch(statement.instruction) do
         Assignment: returnInstructions = createAssignmentFigure(rootNode)
         Parallel: returnInstructions = createParallelFigure(rootNode)
         Pause: returnInstructions = createPauseFigure(rootNode)
         Conditional: returnInstructions = createConditionalFigure(rootNode)
         Goto: store precedingInstructions in GotoMap
   od
   if (statement.instruction != Goto) do
      // draw edges to figures
      for each precedingInstruction in precedingInstructions do
         drawEdge from precedingInstruction to statement.instruction.figure
      od
   od
   precedingInstructions = returnInstructions
   od
   if (exitNode.exists && !precedingInstructions.isEmpty) do
      // draw edges to exit nodes
      for each precedingInstruction in precedingInstructions do
         drawEdge from precedingInstructions to exitNode
      od
      precedingInstructions.clear
   od
   return precedingInstructions
}
```

Listing 5.18. SCG Synthesis – Statement Sequence in Pseudo Code

preceding instruction node for the first statement. For every statement in the sequence the type of the comprising instruction is tested and its corresponding figure is created. Since an instruction may fork off more than one control flow, a list of final instructions is returned. Once a figure is created, the graphical element is mapped to the affiliated instruction and hence is accessible via its statement. If the instruction is a goto, the control flow position is stored in a *goto map* since gotos result in redirections of the control flow and the target figure may not be present yet. The *goto map* is processed after the statement sequence of the SCL program has been evaluated and all instruction figures are present. If it is not a goto, an edge is drawn from every preceding instruction to the newly created figure. Subsequently, the returned instructions from the processed statement are stored as predecessors for the next instruction. Eventually, the statement sequence finishes its iterations. If an exit node is present and there are still remaining instructions in the preceding list, the control flow edges are merged in the exit node. The function returns the remaining instructions in the list for other calling methods. For instance, the conditional instruction calls the createSequenceFigures method to build its branches. Since a conditional has no exit node, the last instructions are returned when the conditional figure is created. Hence, the caller of the conditional has knowledge of not yet merged control flows and may combine them.

5.3.2 Figure Creation

createSequenceFigures, as described in Section 5.3.1, decides which figure elements must be created to represent an SCL program. Consecutively, the corresponding figures must be drawn. Since the creation of a graph figure is mainly of technical nature, only one of these methods is depicted here as an example. The interested reader may find additional information about KLighD in "Transient View Generation in Eclipse" [SSvH12b] presented by Schneider et al. Listing 5.19 illustrated the generation of the assignment figure.

A figure method expects the instruction in question and the parent node which will contain the new figure. The createAssignmentFigure method uses the serializer requested by the synthesis as depicted in Section 5.1.1 to serialize the expression included in the assignment. Secondly, a new graph node element kNode is instantiated. In the case of an assignment it is a rectangular node with default size and thickness. To enhance readability and because the node may be included in a thread hierarchy, which has a gray background, the background of the assignment figure is colored white instead of being transparent. The addNSFixedPorts method adds a north and a south port as connection points for edges to the figure. Since all incoming edges are connected to the north port and all outgoing edges have the south port as source, multiple edges are visually merged automatically by the layouter. The putToLookUpWith method binds the graphical object to a model element in the corresponding editor. This enables the modeller to select figures and find their corresponding representations in the textual model due to their highlighting. Subsequently, the serialized text is added to the figure and the kNode is added to the given parent. The kNode is stored in the instruction mapping before returning. As described in the previous section, the instruction is also returned to the caller of the method for further edge processing.

```
1  def createAssignmentFigure(Assignment assignment, KNode parentNode) {
2      val nodeText = serializer.serialize(assignment)
3      val kNode = assignment.createRectangulareNode(DEFAULT_WIDTH, DEFAULT_HEIGHT)
4          .putToLookUpWith(assignment);
5      kNode.KRendering.add(factory.createKLineWidth.of(DEFAULT_BORDERWIDTH));
6      kNode.KRendering.background = "white".color
7      kNode.addNSFixedPorts
8      kNode.KRendering.add(factory.createKText.of(nodeText).
9          putToLookUpWith(assignment));
10     parentNode.children.add(kNode)
11
12     (assignment as Instruction).addToMapping(kNode, kNode)
13     val returnList = new ArrayList<Instruction>
14     returnList.add(assignment as Instruction);
15     return returnList
16 }
```

Listing 5.19. SCG Synthesis – Assignment Figure Creation

```
kExitNode.data += renderingFactory.createKRoundedBendsPolyline() => [
    it.invisible = true
    it.invisible.modifierId = "de.cau.cs.kieler.scl.klighd.scg.BasicBlockModifier"
];
val bbDataHolder = new BasicBlockDataHolder()
bbDataHolder.SCLProgram = program
bbDataHolder.NodeData = InstructionMapping.clone
bbDataHolder.BasicBlockData.addAll(program.statements.head.getAllBasicBlocks)
kExitNode.data += bbDataHolder
```

Listing 5.20. SCG Synthesis – Basic Block Modifier

5.3.3 Basic Block Modifier Visual Post-processing

Even though this thesis does not illustrate in detail how the visualization of the BBs is implemented since this only comprises the drawing of rectangles and descriptions, this section depicts the invocation of these drawing methods.

Unfortunately, at the time of writing, KLay layered does not fully support hierarchical layouting and also the framework does not provide extension points for general graph post-processing. Nevertheless, a hook for style processing, called *modifier*, exists. The SCG synthesis exploits this hook to add BB information for visualization to the diagram after the layouting completed.

Since any style modifier is invoked automatically subsequent to the layout algorithm, the synthesis creates a new graphical element, a *polyline* in this case, in the last node of the graph, the exit node. The polyline is marked as invisible. As modifier the SCG method responsible for the BB visualization is inserted. Additionally, a new data structure is instantiated to hold the mandatory data. The *data holder class* stores the SCL program, the instruction to figure mapping and all BBs. Lastly, the structure is added to the data field of the exit node where the modifier is able to find it. When the layout algorithm is finished, KLighD calls the modifier which is then provided with sufficient information to draw all BBs.

5.4 Sequentially Constructive Transformations

As illustrated in Section 4.2.2 the transformation of core SCCharts to SCL is made in two steps. Firstly the statechart is translated in an unoptimized form of SCL and then optimized afterwards. The following two sections will describe these transformations in detail.

5.4.1 Core SCCharts to SCL Transformation

The main transformation is structured alongside syntactical elements and follows a top-down approach. It starts on the top-most level, the statechart, and recursively traverses through the hierarchy to translate all encapsulated regions and states. Every

transformation method is called in its designated context and creates a list of new SCL statements for the corresponding statechart element as defined Section 4.1.3. Afterwards, the caller method is responsible for merging generated lists to syntactically correct statement sequences in the SCL model.

The following sub sections explains the methods responsible for the creation of the statement lists of each context: statechart, region, state and transition. To avoid code duplication the transition context is split into two contexts. The first more general context handles the transformation of all outgoing transitions, whereas the *single transition context* is responsible for the code generation of a single transition.

Statechart context

Since Yakindu statecharts always begin with a region, the main state of an SCChart is located in the first region of the statechart in the KIELER extension. As the editor validation automatically checks the model for validity, the transformation method expects a consistent chart. Therefore, there must always be one top level region containing exactly one state which is the main state.

Listing 5.21 depicts the transformation method transformCoreToSCL for transforming core SCCharts to SCL. It requires the statechart in question and an enum set of the type SCLOptimizations, which comprises all SCL optimizations discussed in Section 4.2.3. At first, the function creates a new SCL program instance with the aid of the SCL factory. Secondly, since it assumes a valid statechart, the main state is retrieved and the name of the program is set to the name of the outermost state. Since SCCharts is a KIELER extension to the YSE, it must be cast to a *SyncState*. Also, a valid statechart comprises a section for global variable definitions. A global SCL variable is introduced for every declaration found in the state. The definition is generated directly out of the declaration

```
1  def Program transformCoreToSCL(Statechart statechart,
2     EnumSet<SCLOptimization> optimizations) {
3     val targetProgram = SclFactory::eINSTANCE.createProgram();
4     val mainState = statechart.regions.get(0).vertices.get(0) as SyncState
5     targetProgram.setName(mainState.name)
6
7     // Add all declarations of the main state to the declaration of the program.
8     for(declaration : mainState.scopes.get(0).declarations) {
9       targetProgram.definitions.add(createVariableDefinition(declaration));
10    }
11    // Create a list of statements for the main state
12    // (and all including regions and states)
13    // and add them to the program.
14    var statements = transformCoreStateToSCL(mainState, optimizationFlags)
15    targetProgram.statements.addAll(statements)
16
17    targetProgram
18 }
```

Listing 5.21. SCL Transformation – Statechart Context

in the statechart via the SCL create extension. Finally, the transformation method for states can be invoked with the main state as a parameter. The generated sequence of statements is added to the program before returning to the caller and ultimately represents the complete SCL translation.

Region Context

To translate a region the transformation method transformCoreRegionToSCL, shown in Listing 5.22, needs the actual region object and the optimization flags as parameters and returns a list of statements, which embody the SCL code of the corresponding region. At first the create extension generates a new empty list of statements. Afterwards, all SyncStates are copied and sorted by state type accordingly to ensure that initial states are processed first and final states come last. Subsequently, the state transformation is invoked for every comprising state. To facilitate optimizations the list of statements for each state is stored in a temporary array statesStatements. Afterwards, if the stateposition optimization is chosen, the statement lists are realigned before further optimizations are invoked. Finally, if selected, all superfluous goto instructions and labels are removed and the generated merged list is returned.

```
1  def List<Statement> transformCoreRegionToSCL(Region region,
2      EnumSet<SCLOptimization> optimizations) {
3      var newStatements = createNewStatementList()
4      // List of all states in this region.
5      val states = ImmutableList::copyOf(region.getVertices.
6          filter(typeof(SyncState))).sort(e1, e2 | compareSCLRegionStateOrder(e1, e2))
7
8      // In order to execute the state position optimization
9      // all statement lists are stored in an array.
10     var statesStatements = new ArrayList<ArrayList<Statement>>
11     for (state : states) {
12         statesStatements.add(transformCoreStateToSCL(state, optimizations))
13     }
14     // If selected execute the state position optimization.
15     if (optimizations.contains(SCLOptimization::STATEPOSITION)) {
16         statesStatements = statesStatements.optimizeStatePosition
17     }
18     // Add all statements to the new list.
19     for(stateSet : statesStatements) {
20         newStatements.addAll(stateSet)
21     }
22     // Run optimizations if selected.
23     if (optimizations.contains(SCLOptimization::GOTO))
24         newStatements = newStatements.optimizeGoto
25     if (optimizations.contains(SCLOptimization::LABEL))
26         newStatements = newStatements.optimizeLabel
27
28     newStatements
29 }
```

Listing 5.22. SCL Transformation – Region Context

```
def List<Statement> transformCoreStateToSCL(SyncState state,
    EnumSet<SCLOptimizations> optimizations) {
    var newStatements = createNewStatementList()
    val stateID = state.getHierarchicalName()
    val emptyStatement = createSCLEmptyStatement
    emptyStatement.label = stateID
    newStatements.add(emptyStatement)
    if (state.isComposite()) {
        // If there is only one region,
        // the code of that region can be added without further processing.
        // If there are more regions, a parallel statement has to be created.
        if (state.getRegions().size<2) {
            var regionInstructions =
                transformCoreRegionToSCL(state.getRegions().head, optimizations)
            newStatements.addAll(regionInstructions)
        } else {
            var parallel = SclFactory::eINSTANCE.createParallel();
            for(stateRegion : state.getRegions()) {
                val regionInstructions = transformCoreRegion(stateRegion,
                   optimizations)
                parallel.getThreads().add(createSCLThread(regionInstructions))
            }
            newStatements.add(parallel.createStatement)
        }
        // Process normal terminations
        // and add assignment statements for every transition trigger.
        if (state.getNormalterminations.size>0) {
            val transition = state.getNormaltermiantions.head
            val targetState = transition.getTarget as SyncState
            val effect = transition.getEffect()
            if (effect != null)
                newStatements.addAll(createSCLAssignments(effect).createStatements)
            var goto = createSCLGoto(targetState.getHierarchicalName())
            newStatements.add(goto.createStatement)
        }
    } else if (!state.isFinal) {
            // The state is not a composite or final state.
            // Add statements of the transitions transformation.
            newStatements.addAll(state.
               transformStateTransitionsToSCL(optimizations))
        }
    }
    newStatements
}
```

Listing 5.23. SCL Transformation – State Context

State context

Listing 5.23 exemplifies the transformation method for state, transformCoreStateToSCL. As expected, it requires a SyncState and the set of selected optimizations. At the beginning of the function a new *ID* for the state is generated via getHierarchicalName in the SCL naming extension. It is comprised of the names of all superior hierarchies separated by an underscore. If a state or region does not own a unique name, its object hash code is used instead to make the ID exclusive. Then, an empty statement is instantiated to hold

a starting label. This label marks the beginning of the state source code and is the entry point for goto jumps originating in other states. The ID of the state is used as a label identifier.

If the state is a composite state, it may contain an arbitrary number of regions. In the case of one region the resulting SCL code of the included compartment can be inserted directly without further processing. However, if the state comprises more than one region, it creates a concurrent context and an SCL parallel statement, parallel in line 17, must be created with aid of the SCL factory. For each region the source code is generated and added as the statement sequence of a new thread in the parallel statement. Subsequently, the newly built concurrent construct is added to the statement list of the state. To close the processing of composite states, any normal termination code must be assembled. Although normal terminations do not possess a trigger they may hold an effect including an arbitrary number of actions. If a transition includes an effect, the createSCLAssignments will construct SCL assignments for every comprised action in the effect. Eventually, the code for a normal termination finalizes with a goto jump to the target state for the transition. createSCLGoto produces a new SCL goto statement and sets the passed parameter as target label.

If the state is not a composite or final state, the transformation method for state transitions is called. If it is a final state, only the label created at the beginning is returned.

Transitions context

As mentioned in the introduction to this section, the handling of transitions is split in two contexts to avoid code duplication. The *Transitions Context* is responsible for processing all outgoing transitions of a state whereas the code generation for a single transition is the task of the *Single Transition Context* described in the following sub section.

Since the transitions transformation transformStateTransitionsToSCL is relatively large, it is depicted in pseudo code in Listing 5.24. As usual, it creates a new statement list and gathers data about the different types of transitions in transitions and immediateTransitions. To process a single transition the subsequently discussed method transformTransitionToSCL can be called upon.

Firstly, all immediate transitions are translated to SCL. A state is transient if it does not consume any time and therefore does not need a pause instruction in SCL. Two conditions must be met for a state to be transient. At first, all outgoing transitions must be immediate and secondly, at least one transition must be active in any tick. A transition without trigger is called *default transition*. It is active in each tick, provided no superior transition activates beforehand. In the listing, if only immediate transitions are present, the method checks for a default transition. If none is found, a pause and a goto to the state itself are created. This represents an implicit self-loop. If selected, optimizeSelfLoop will perform the self-loop optimization discussed in Section 4.2.3. Afterwards the transformation method exits, since no non-immediate transitions

```
1  def List<Statement> transformStateTransitionsToSCL(SyncState state,
2     EnumSet<SCLOptimizations> optimizations) {
3     newStatement = createNewStatementList
4     transitions = state.outgoingTransitions.filter(typeof(SyncTransition))
5     immediateTransitions =
6       ImmutableList::copyOf(transitions.filter(e | e.isImmediate))
7     // Invoke transition transformation for each immediate transition
8     for each transition in immediateTransitions do
9        newStatements.addAll(transition.transformTransitionToSCL)
10    od
11    if immediateTransitions = transitions do
12       // Transient state?
13       if !state.hasDefaultTransition do
14         // Add pause and self-loop, if no default transition is present
15        newStatements.add(SCLPause)
16        newStatements.add(SCLGoto(state))
17       od
18      call optimizeSelfLoop(newStatements)
19      return newStatements
20    od
21    // Insert pause
22    newStatements.add(SCLPause)
23    // Process all transitions
24    for each transition in transitions do
25       newStatements.addAll(transition.transformTransitionToSCL)
26    od
27    call optimizeDuplicateTransitions(newStatements)
28    if !state.hasDefaultTransition do
29       newStatements.add(SCLGoto(state))
30    od
31    return newStatements
32 }
```

Listing 5.24. SCL Transformation – Transitions Context in Pseudo Code

are present.

If non-immediate transitions are present, the state consumes time and therefore, a pause statement is mandatory. Then, all SCL code for transitions is generated. This includes the previously processed immediate transitions, due to the fact to interleaving priorities between the two types of transitions. Subsequently, the transition code optimization, also described in Section 4.2.3, is invoked when chosen. Similar to the transient case, if no default transition is present, a goto to the state itself is created to model the implicit self-loop.

Single transition context

Finally, the construction of code for a single transition is relatively straightforward. As illustrated in Listing 5.25 transformStateTransitionToSCL retrieves the target of the transition and checks for a trigger. If a trigger is present, an SCL conditional with potential effect code is created and added to the list of statements. Otherwise, the effect code can be inserted directly followed by a goto since no condition is to be evaluated.

```
def List<Statement> transformStateTransitionToSCL(SyncTransition transition) {
    var newStatements = createNewStatementList()
    // Create the goto statement the transition points to
    var targetGoto = createSCLGoto(transition.target.getHierarchicalName())
    // Transition trigger and effect
    val transitionAssignments = createSCLAssignment(transition.effect)
    // If a trigger is present, add a conditional statement.
    if (transitionTrigger.trigger.exists) {
        var conditional = createSCLConditional(transitionTrigger)
        // Add the assignment statements to the conditionals statement sequence.
        if (transitionEffect.exists)
            conditional.statements.addAll(transitionAssignments.createStatements)
        conditional.statements.add(targetGoto.createStatement)
        newStatements.add(conditional.createStatement)
    } else {
        // If there is no trigger, simply add the assignments and a goto statement.
        newStatements.addAll(transitionAssignments.createStatements)
        newStatements.add(targetGoto.createStatement)
    }
    newStatements
}
```

Listing 5.25. SCL Transformation – Single Transition Context

5.4.2 SCL Optimizations

The following sections describe the implementation of the SCL code optimizations mentioned in Section 4.2.3 and used in the implementation of the SCL transformations, Section 5.4.1.

Self-loops Optimization

Only states with one outgoing transition and a self-loop are eligible for this kind of optimization. However, states with implicit self-loops are also considered since the transformation in Section 5.4.1 constructs the loop for them. They are structured as depicted in Listing 5.27 when entering the optimization method optimizedSelfLoop, seen

```
def List<Statement> optimizeSelfLoop(List<Statement> originalStatements,
   List<Statement> stateStatements) {
   var newStatements = createNewStatementList()
   var conditional = originalStatements.head.getInstruction as Conditional;
   val newConditional = createSCLConditional()
   newConditional.expression = conditional.expression.negate
   newConditional.statements.addAll(stateStatements)
   newStatements.add(newConditional.createStatement)
   newStatements.addAll(conditional.statements)
   newStatements
}
```

Listing 5.26. SCL Optimization – Self-loop

```
1 _stateA:
2 if I then
3     goto _stateB
4 end;
5 goto stateA;
6 _stateB:
```

Listing 5.27. SCL Self-loop Example

```
1 _stateA:
2 if !I then
3     goto _stateA
4 end;
5 goto stateB;
6 _stateB:
```

Listing 5.28. Optimized Self-loop Examp.

in Listing 5.26. Firstly, a new conditional is created and the old expression is copied and negated. Notably, rather than returning a double negation, negate will eliminate a present one when invoked. The new conditional is then filled with the previous statements of the state while the instructions of the obsolete conditional serve as the new statement list of the state itfself. Essentially, the two lists of statements swap their positions with the expression in the conditional being negated. An example is illustrated in Listing 5.28. The new instruction ordering facilitate the next two optimizations.

Goto Optimization

As stated in Section 4.1.3 a goto must not jump out of or into a concurrent compartment. Therefore, the goto and label optimizations are used in a region context since all jumps included in a region must reference labels which are also contained in that specific region.

```
1  def ArrayList<Statement> optimizeGoto(List<Statement> statements) {
2     val newStatements = createNewStatementList()
3     // Iterate through all statements
4     for(int i: 0..(statements.size - 1)) {
5        // Since xtend does not know a common next or continue instruction
6        // we are going to remember which instruction must be skipped.
7        var boolean skip = false
8        val statement = statements.get(i)
9        // if the statement is a goto statement and has succeeding statements...
10       if (statement.hasInstruction && statement.instruction instanceof Goto &&
11          i < statements.size - 1) {
12          // ... search the next statement.
13          var nextStatement = statements.get(i + 1)
14          // If the succeeding instruction statement is the target statement
15          // of the goto jump, mark it as 'to be skipped'.
16          if (nextStatement != null &&
17             nextStatement.getLabel == statement.getInstruction.asGoto.targetLabel)
18                skip = true
19       }
20       // Add all statements not marked as 'to be skipped'.
21       if (!skip)
22          newStatements.add(statement.copy)
23    }
24    newStatements
25 }
```

Listing 5.29. SCL Optimization – Goto

s OptimizeGoto, listed in Listing 5.29, removes any goto instruction which targets a directly successive label. Such a jump is superfluous, since the control flow proceeds along that path anyhow. The method iterates through the given list of statements and searches for gotos. If a following statement, denoted as nextStatement, is a label the goto can be removed. Since Xtend does not support a *continue* instruction, statements which must be left out are marked by the skip variable. It is set, if a succeeding statement exists and the label of that statements equals the target label of the goto. GetLabel returns *null*, if the statement does not comprise a label.

Label Optimization

Listing 5.30 depicts the optimization for labels. Any label which is not referenced by a goto can be eliminated. Therefore, the label optimization amplifies the gain of the previous goto upgrade. Xtend provides functions to solve this elegantly. In a first step optimizeLabel queries all goto instructions a given statement list comprises. Subsequently, all statements which are neither an empty statement nor referenced by a goto are added to a new statement list. By this, any not referenced label instruction is removed.

```
1  def ArrayList<Statement> optimizeLabel(List<Statement> statements) {
2      val newStatements = createNewStatementList()
3      // Copy all goto statements.
4      val gotos =
5          ImmutableList::copyOf(statements.getAllContents.filter(typeof(Goto)))
6      // And filter all empty label statements that are not referenced by any goto.
7      newStatements.addAll(statements.filter(e |
8          !(e instanceof EmptyStatement) ||
9          (gotos.exists(f | f.targetLabel == (e as EmptyStatement).label)
10     )))
11     newStatements
12 }
```

Listing 5.30. SCL Optimization – Label

State Positions Optimization

To further facilitate the goto and label optimizations optimizeStatePositions tries to rearrange the states of a region to build matching goto and label combinations. This upgrade must be executed before the invocation of the two previously discussed optimizations. OptimizeStatePositions expects a list of statements lists. Each sublist represents the code of a single state. If a state closes with a goto instruction, it is checked whether or not another state starts with a corresponding label. Subsequently, the state of the actual iteration is added to the new statement list. If a matching state is found, its state source code is added directly behind the actual state, provided it was not already processed. Otherwise, the code of the original state is added to the new list without alteration.

```
1  __S:
2  if T1 then
3     goto __Sf;
4  end;
5  if T3 then
6     goto __Sf;
7  end;
8  pause;
9  if T1 then
10    goto __Sf;
11 end;
12 if T2 then
13    goto __Sf;
14 end;
15 if T3 then
16    goto __Sf;
17 end;
18 goto __S;
19 __Sf:
```

Listing 5.31. SCL Duplicate Transition

```
1  __S:
2  if T1 then
3     goto __Sf;
4  end;
5  if T3 then
6     goto __Sf;
7  end;
8  pause;
9  if T1 then
10    goto __Sf;
11 end;
12 if T2 then
13    goto __Sf;
14 end;
15 goto __S;
16 __Sf:
```

Listing 5.32. Optimized Dup. Transition

Duplicate Transitions Optimization

The elimination of duplicated transitions is made in a straightforward manner. Subsequent to the naive SCL source generation of a state any transition code in the *depth* of that state can be removed if it is called again in the *surface* and is executed in the same order.

Listing 5.31 shows the generated SCL code of a state with three transitions, $T1$, $T2$ and $T3$ with priorities 1, 2 and 3. $T1$ and $T3$ are immediate and are hence checked before the pause instruction. Subsequently all three transitions are tested again. The optimization detects $T3$ as being the last conditional checked in both the surface and the depths and removes it. However, it can not eliminate $T1$ in the depth, since even in the depth the conditional of $T1$ must be evaluated before $T2$. Thus, $T1$ is tested twice in consecutive ticks, but the code stays correct. The optimized SCL code is shown in Listing 5.32.

5.4.3 The Sequential Tick Function

As explained in Section 4.2.7 subsequent to the analyses performed on a generated SCL program the desired generic tick function can be engineered, provided that the program is *ASC-schedulable*. The tick function is generated out of the BBs and only comprises *guard* calculations and assignments nested in guarded conditionals. Therefore, all pause and parallel statements are omitted.

The transformation is done in two steps. At first it must be decided which BBs are eligible to be processed and secondly, generate the SCL code for a particular BB when all of its preconditions are met.

Basic Blocks Arrangement

The first part is included in the main transformation method transformSCLToSequentialSCL, depicted in Listing 5.33. At the beginning of the routine the target program is instantiated. All definitions of the source program are copied since they comprise the variable interface and the initial GO signal is added. Then, all BB are queried. For each BB a variable for its guard is created. If the BB is the surface of a pause, a corresponding _pre definition is also added. Furthermore, if the BB is a join of a parallel statement, variables for the synchronizer's empty flags generated.

The iteration for the BB arrangement subsequently follows. For this, all BB are copied into an new list basicBlockPool. The while loop iterates as long as basicBlockPool is not empty. In each iteration the flag poolAltered signals whether or not a BB was processed in this turn. Then, the basicBlockPool is copied again to the temporary list tempPool. This list is used to check for eligible BB placements in this turn.

For each BB basicBlock in the temporary pool the boolean ready flags the placement status of the actual block. While testing a block, each predecessor of that block that is not a PauseSurface is examined. If it is still in the basicBlockPool it has not been placed yet and at least one precondition is not met. Wherefore, the ready flag is set to false. If the BB is a ParallelJoin, these requirements must also be tested for the blocks needed to evaluate the *empty flags* of the join synchronizer. In this context, stripSurface removes any surface dependencies in BBs contained in threads. This is necessary since a feedback edge from a join to a corresponding fork node may be instantaneous. In this case cyclic dependencies are broken inside the parallel statement. Again, if at least one mandatory block still persists in the basicBlockPool, the guard can not be evaluated. Following this, an analogous check is performed for data dependencies the block inherits from its statements.

Similar to the calculation of the guards depending on the predecessors of a given block, the guards of data dependencies must be evaluated beforehand. Otherwise the guard expression can not be calculated due to missing results of required guards. Eventually, if the ready flag is still true, all preconditions are met and the transformation method for the BBs, transformBasicBlock, can be invoked to insert the block at the actual position in the targetProgram. The BB is then removed from the pool and other precondition tests will recognize this block as being processed. Therefore, it may now guard succeeding BBs. Additionally, poolAltered is set to true.

If poolAltered is still false subsequent to a complete precondition test iteration, no remaining BB meets its requirements. In such a case a fix point is reached and the program is not Acyclic Sequentially Constructive (ASC)-schedulable. Therefore, a correlative exception NotASCSchedulableException is thrown and the calculation terminates.

The placement of BBs continues as long as there are still blocks in the pool or until no further block can be positioned. Finally, if all BBs are in place, the method creates assignments to transfer the guard values of surfaces to the corresponding *_pre* guards for the consecutive tick. Then, the newly engineered sequential SCL program is returned.

```
def Program transformSCLToSequentialSCL(Program program) {
   targetProgram = SCL.createProgram
   targetProgram.definitions.addAll(program.definitions)
   targetProgram.defintions.add(createVariableDefintion('GO'))
   // Copy definitions and create guard variables
   basicBlocks = program.statements.head.getAllBasicBlocks
   for each basicBlock in basicBlocks do
      targetProgram.defintions.add(createVarDef(basicBlock.guardName)
      if (basicBlock.isPauseSurface) do
         targetProgram.defintions.add(createVarDef(basicBlock.guardName + '_pre')
      if (basicBlock.isParallelJoin) do
         targetProgram.definitions.addAll(createVarDef(basicBlock.emptyFlags)
   od
   // Basic Block preconditions test
   basicBlockPool = basicBlocks
   while not basicBlockPool.empty do
      poolAltered = false
     tempPool = basicBlockPool.copy
     for each basicBlock in tempPool do
        ready = true
       // Check predecessors
       for each predecessor in basicBlock.getPredecessors do
          if not predecessor.isSurface do
            if basicBlockPool.contains(predecessor) do ready = false
            if basicBlock.isParallelJoin do
               joinGuards = predecessor.getBasicBlocks.getSurfaces
              if basicBlockPool.contains(joinGuards) do ready = false
            od
         od
       od
       // Check data dependencies
       for each dependency in basicBlock.getDependencies do
          if not dependency.isSurface and basicBlockPool.contains(dependency)
            ready = false
         // Place transformed block and remove it from the pool
       if ready do
          targetProgram.statements.addall(transformBasicBlock(basicBlock))
         basicBlockPool.remove(basicBlock)
         poolAltered = true
       od
     od
     if not poolAltered do
        throw NotASCSchedulableException
   od
   // Transfer surface values
   for each basicBlock in basicblocks do
      if basicBlock.isSurface do
         targetProgram.statements.add(
            createAssignment(basicBlock.guardName + '_pre', basicBlock.guardName))
   od
   return targetProgram
}
```

Listing 5.33. Sequential SCL – Main Loop

Guards Generation

Section 4.2.6 defines the guard expressions necessary to evaluate the actual state of any guard. Furthermore, Section 4.2.7 describes the generation of the sequential tick function utilizing the guards to evaluate the control flow of the SCL program.

transformSCLToSequentialSCL calls transformBasicBlock, shown in Listing 5.35, to generate the guard source code. The method creates the expression for the guard and adds a conditional including any code for assignments if necessary. In the case of ParallelJoin it also creates the join synchronizer.

The method starts again with the creation of a new statement list. Additionally, all predecessors are queried. If the BB is in fact a join, the first if-block is executed and a synchronizer as discussed in Section 4.2.6 is constructed. It creates two expressions. The terminationExpression builds the condition that at least one thread must be exited in this tick instance. The synchronizerExpression will hold the concatenated expressions of the

```
1  def List<Statement> transformBasicBlock(Program program) {
2     newStatements = createNewStatementList
3     predecessors = basicBlock.getPredecessors
4     if basicBlock.isParallelJoin do
5        // Generate synchronizer
6        synchronizerExpression = createNewExpression
7       terminationExpression = createNewExpression
8        for each predecessor in predecessors do
9          // Build empty flag for each thread
10         emptyExpression = createNewExpression
11        subExpression = createNewExpression
12        guards = predecessor.getBasicBlocks.getSurfaces
13        emptyExpression = createOrConcatenation(emptyExpression, guards).negate
14        for each guard in guards do
15           if guard.reachesExitNode do
16             exitExpression =
17                createAndConcatenation(guard.guardName,
18                  guard.conditionalExpression)
19             terminationExpression =
20                createOrConcatenation(terminationExpression, exitExpression)
21             subExpression = createOrConcatenation(subExpression, exitExpression)
22          od
23        od
24        newStatements.add(createEmptyAssignment(emptyExpression))
25        subExpression = createOrConcatenation(subExpression, emptyExpression)
26        synchronizerExpression =
27           createAndConcatenation(synchronizerExpression, subExpression)
28     od
29     // Finalize guard for the synchronizer
30     synchronizerExpression =
31        createAndConcatenation(synchronizerExpression, terminationExpression)
32     newStatements.add(createGuardAssignment(basicBlock.guardName,
33        synchronizerExpression))
34     return newStatements
35  od
```

Listing 5.34. Sequential SCL – Basic Block Transformation

```
36      // No parallel basic block
37      guardExpression = createNewExpression
38      if basicBlock.isInitialBlock do
39         // Add GO signal
40         guardExpression = createElementReference('GO')
41      for each predecessor in predecessors do
42         // Build guard
43         activator = createElementReference(predecessor.guardName)
44        if predecessor.isConditionalPredecessor do
45           if predecessor.isInTrueBranch do
46             activator = createAndConcatenation(activator,
47               predecessor.conditionalExpression)
48          od else do
49             activator =
50              createAndConcatenation(activator,
51                 predecessor.conditionalExpression.negate)
52          od
53        od
54         guardExpression = createOrConcatenation(guardExpression, activator)
55        newStatements.add(
56           createGuardAssignment(basicBlock.guardName, guardExpression))
57        // Build conditional for scl assignments
58        guardConditional = createNewConditional
59        for each statement in basicBlock.statements do
60           if statement.isAssignment do
61             newStatement = statement.copy.transformReferences
62          od
63        od
64        if not guardConditional.statements.empty do
65           newStatements.add(guardConditional)
66      od
67      return newStatements
68  }
```

Listing 5.35. Sequential SCL – Basic Block Transformation (cont.)

preceding *empty flag* evaluations connected with the terminationExpression. It therefore represents the guard of the join.

In the parallel join context predecessors holds the starting BBs in the preceding threads. For each thread an emptyExpression is instantiated. Simultaneously one sub expression for the synchronizerExpression, denoted subExpression, is built. All enclosed surfaces in the threads are queried and stored in guards. Subsequently, they are added to the emptyExpression. If the block reaches the exit node of its thread, it is concatenated with the expression of a conditional statement if the guard depends on an environment variable, and then added to the terminationExpression and the subExpression. Before proceeding to the next guard, the emptyExpression is added to the SCL code and subExpression linked to the synchronizerExpression. Subsequent to the guard iteration the expression of the synchronizer is concatenated with the termination precondition of the threads and added as guard to the list of statements. As the calculation of the join guard finishes here, the method is left at this point.

If the BB is not a join of concurrent threads, the evaluation of its guard is less complex.

The guard is constructed in the guardExpression. If the block is an initial BB, it depends on the *GO* signal of the environment. For each predecessor an expression representing the incoming activator is generated. It is constructed frin the REFEXP of the predecessor block and connected to the expression of a conditional statement when present. If the BB persists in the *else branch* of the conditional, the conditionalExpression is negated. Each activator is then added to the guardExpression via createOrConcatenation. Subsequently, the new guard is created with createGuardAssignment and added to the SCL code.

Finally, a new guardConditional has to be created to encapsulate all assignments of a BB. The can simply be copied but the reference of the ASSEXP must be adjusted to the variable definition of the target program. This is done in transformReferences. If at least one assignment statement exists, the resulting conditional list will not be empty and is therefore added to the list of statements. The method eventually returns.

Chapter 6

Experimental Results

To validate the results and draw conclusions the code generation approach was evaluated in KIELER. KIELER provides a simulation environment based on the abstract language S and already comprises S transformations for SyncCharts. To simulate generated SCL code in KIELER a transformation to translate SCL to S was developed. Since SyncCharts can be converted to SCCharts the simulation results of equivalent models are comparable. However, S uses signals as communication mechanism itself and therefore might favour communications in SyncCharts over the variable concept implemented in SCCharts.

Listing 6.1 depicts the SCL code of a simple tick function example with three guards. The corresponding S program is shown in Listing 6.2. As illustrated the guard assignments are translated to signal emissions again and hence might interact more efficient with the SyncChart instead of the SCL approach.

```
1  module Simple_tick
2  output boolean O1, O2, O3;
3  boolean GO, g0, g1, g1_pre, g2;
4  {
5     g0 = GO;
6     if g0 then
7        O1 = false;
8     end;
9     g2 = g1_pre;
10    if g2 then
11       O2 = true;
12    end;
13    g1 = g0 || g2;
14    if g1 then
15       O3 = true;
16    end;
17    g1_pre = g1;
18 }
```

Listing 6.1. Tick Function in SCL

```
1  synchronous program Simple_tick ( 1 )
2
3  output signal O1, O2, O3: boolean;
4  signal GO, g0, g1, g1_pre, g2: boolean;
5
6  state( _go ) {
7     emit(GO(true));
8     trans(_tickStart);
9  }
10
11 state( _tickStart ) {
12    emit(g0(?GO));
13    if ( ?g0 = true ) {
14       emit(O1(false));
15    };
16    emit(g2(?g1_pre));
17    if ( ?g2 = true ) {
18       emit(O2(true));
19    };
20    emit(g1(?g0 or ?g2));
21    if ( ?g1 = true ) {
22       emit(O3(true));
23    };
24    emit (g1_pre(?g1));
25    pause();
26    trans(_tickStart);
27 }
```

Listing 6.2. Tick Function in S

Although S might be suited better to translate SyncCharts, it can be used to compare the two approaches regarding *execution time* and the *size of the resulting binary code* since the signal emissions are translated to C macros and expanded during the compilation. The evaluation focuses on the differences in the execution times.

Test Set-up

Two different kinds of tests were executed to compare the results of the different approaches. The first analysis takes randomly generated statecharts and compares the execution times and executable sizes as state and hierarchy layer count increases. The second evaluation tests small common examples and interprets the results for these. The last section describes observations regarding SCL guards.

All tests were done on an Intel Core^{TM}i7-3770, 3.4 GHz with 16 GB RAM running a 64-bit Windows 7^{TM}. Each simulation run comprises 100 macro steps and was sampled five times to calculate the mean values. As input data KIELER generated .eso files filled with random input value occurrences. .eso file are esterel trace files that can also be used as input data in KIEM simulations. In every test the same input traces were used for both models. Summarized, each statechart was simulated for 500 macro steps and both approaches used the same input data for their simulations.

Figure 6.1 depicts the preparation chain for each test.

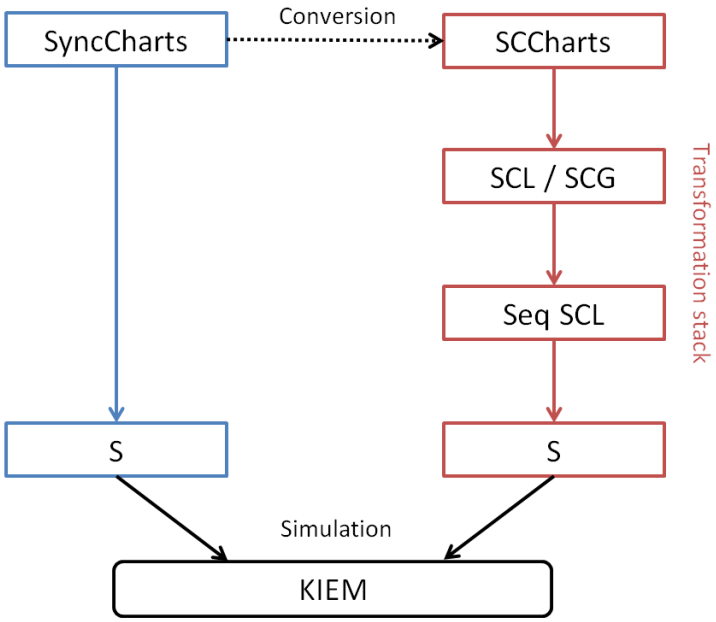


Figure 6.1. Test Set-up – Transformations

6.1 Scaling Approach Evaluation

To measure the execution times in statecharts of different sizes, a series of random SyncCharts were created and converted to SCCharts and subsequently to SCL. Finally, both were translated to S and simulated with KIEM. The first series increases the count of states in every run whereas the second added additional hierarchy layers.

State Count Growth Comparison

Ten statecharts with increasing state count were created for this simulation. The models comprise an average transition count of two and no hierarchy. The interface consists of three input and three output signals or variables.

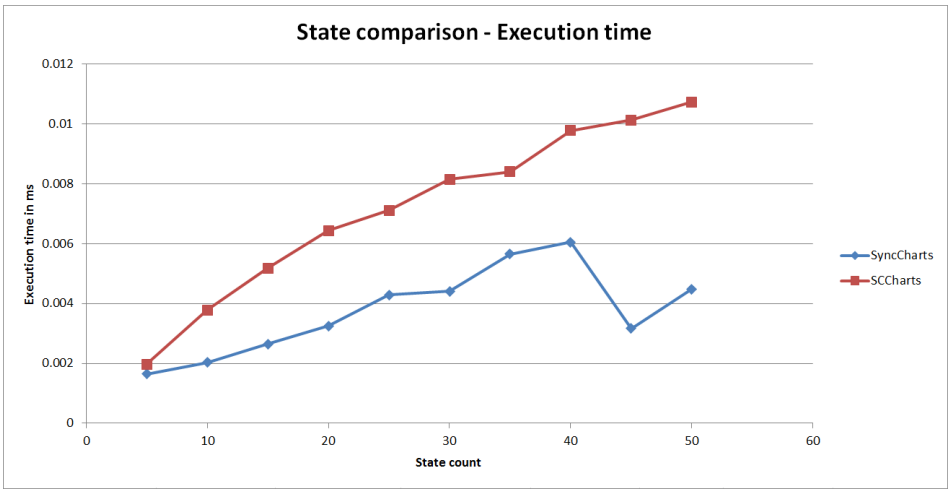


Figure 6.2. State Comparison – Execution Time

Figure 6.2 shows the execution time compared to the count of states. Although both types of charts pursue an increasing global trend, the growth in SCL is more constant and higher as in SyncCharts in the mean. Furthermore, the local execution time of SyncCharts is not only bound to the size of a chart but depends highly on its structure and the input data. Therefore, execution times can be lower in larger charts as seen in the figure. Actually, the random nature of the models and the input data were seen to have a more unpredictable impact on the timing of the SyncChart simulation.

The size of the compiled source codes is depicted Figure 6.3. Its growing as the state count increases and SCL has slightly smaller executables than SyncCharts.

Hierarchy Layer Growth Comparison

As setup for this simulation five SyncCharts of different hierarchy depth and their corresponding SCCharts were created automatically. Every macro state comprises two

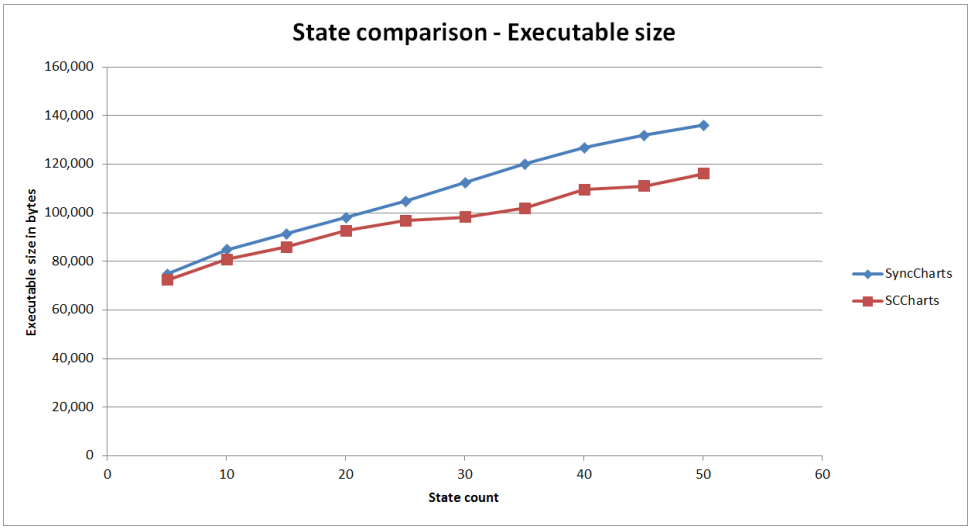


Figure 6.3. State Comparison – Executable Size

parallel regions with five states each. In every test the count of hierarchy layers is increased and each new layer also includes the starting setup, two regions and five states each. Again, the average transition count is two, but each hierarchy layer expects its own signal interface, therefore, the size of the interface increases as the layer count increases. The same is true for variables in the SCChart variant.

The test at hierarchy layer 3.5 was introduced to validate the large increase in the execution time in the SyncChart simulation. In this case only one parallel region has a hierarchy depth of four. The second region remains at three.

Figure 6.4 illustrates the execution times of diagrams with different hierarchy depths. Although SCL starts above SyncCharts similar to the state count comparison, the execution time of the SCLs increases more steadily. However, the execution time of SyncCharts rises drastically after a layer depth of three is reached.

Similar to the observations made in the state count evaluation, the timing of SyncCharts highly depend on the structure of the chart and the input data whereas SCL pursues a more linear increase. To visualize this further, Figure 6.5 exemplifies the variations in the execution time of the hierarchy test with a depth of four in one test run.

The timing of SyncCharts fluctuates immensely depending on the state of the SyncChart and its actual input data whereas SCL proceeds steadily with the same input data given.

The differences in the executable size behave similar to the observations made in the state count evaluation with SCL's executables being a little smaller than the SyncCharts' executables.

As observed till now, the quality of the results depends on the structure of the charts and the input data used in the simulation whereas SCL being a little slower

Figure 6.4. Hierarchy Layer Comparison – Execution Time

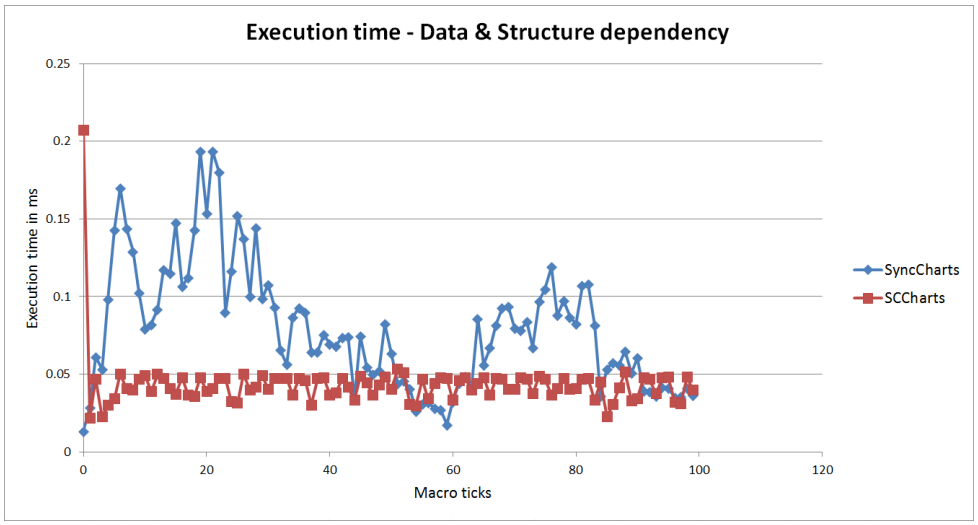


Figure 6.5. Execution Time Comparison at Hierarchy Test with Depth Four

than SyncCharts as long as in lower hierarchy depths but more robust regarding timing fluctuations in connection with variations in the chart structure and in the input data.

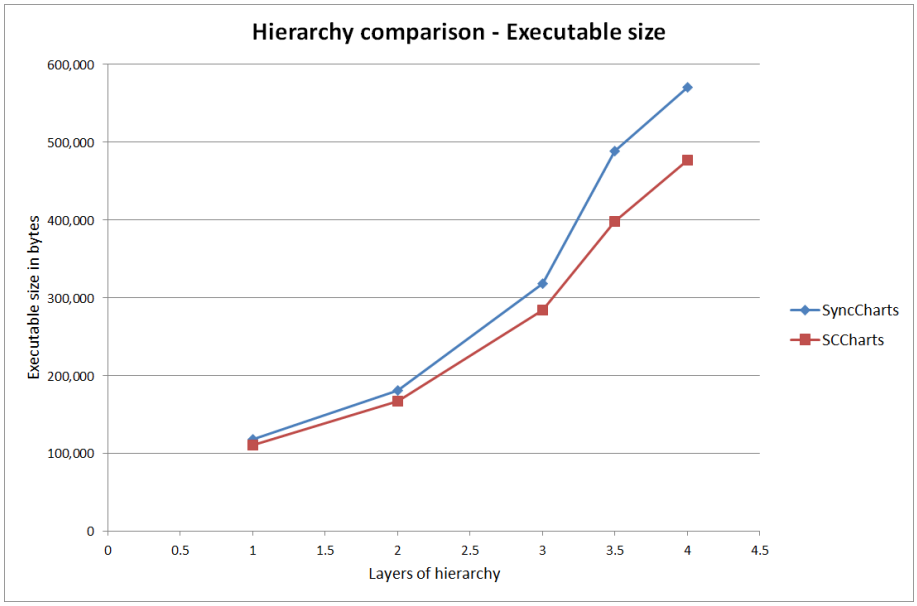


Figure 6.6. Hierarchy Layer Comparison – Executable Size

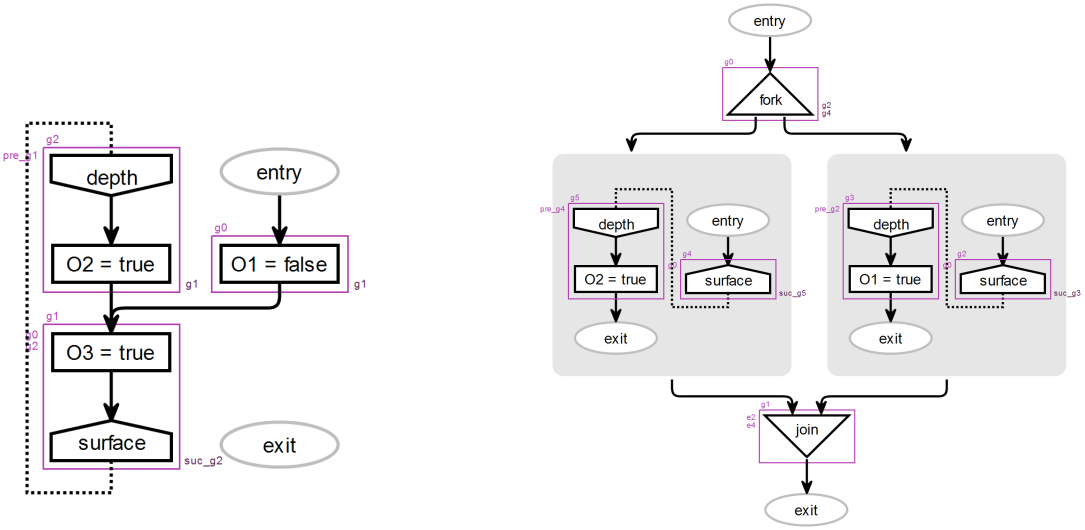
6.2 Common Example Evaluation

To not depend on random Statecharts only this section compares four common examples of reasonable sizes. The first two examples, Simple and SimpleConcurrency are small statecharts depicted in Figure 6.7. They serve as minimal test cases for the transformation without and with concurrency. The third example is ABSWO-xp, a variant of ABRO, transformed to an core chart free of pre-emptions. It was presented in "Sequentially Constructive Concurrency - A conservative extension of the synchronous model of computation" [vHMA+13b]. The last test comprises the shared resource example, was presented by Edwards in "Tutorial: Compiling Concurrent Languages for Sequential Processors" [Edw03]. It illustrates a producer-consumer example in Esterel and was translated to SyncCharts and SCL.

In the Simple case SCL acts a bit faster due to the fact that it has only to evaluate three guards. Including the pause register only four assignments must be executed in each tick whereas SyncCharts traverses through the structure of the statechart.

For exactly the same reason SCL performs worse in the next test, SimpleConcurrency. Although the statechart is similarly small as Simple, the concurrent component results in the creation of a synchronizer and doubles the assignments needed to evaluate the tick function.

The code of the synchronizer has not such a big impact on the execution time in the relatively larger examples since it comprises only a fraction of the guards. SCL performs quite well in ABSWO-xp and even better in the shared resource example since



(a) Simple Example **(b)** SimpleConcurrency Example

Figure 6.7. Common Examples

the generated code contains less guards. Unlike ABSWO-xp which translates to 26 guards in this implementation, the shared resource example only needs 20.

The sizes of the compiled code are almost the same with SCL being slightly smaller which supports the observations of the evaluations in Section 6.1.

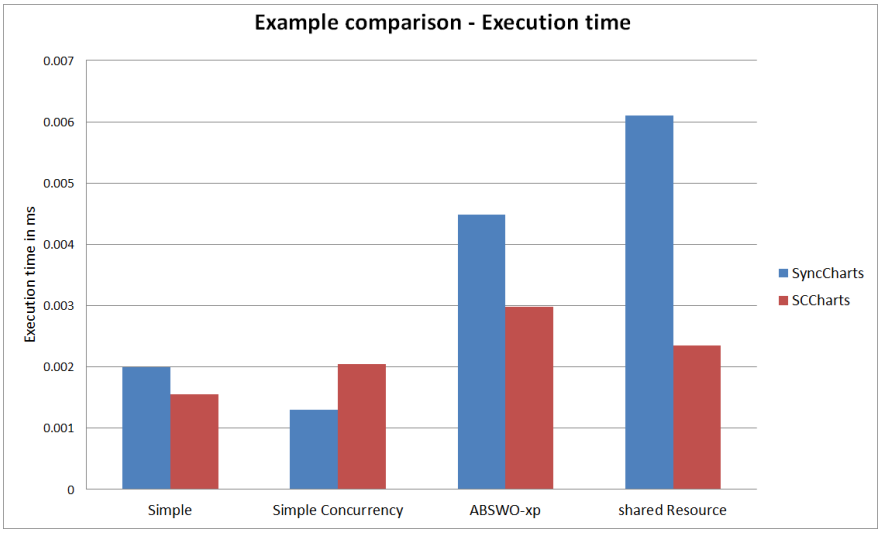


Figure 6.8. Common Example Comparison – Execution Time

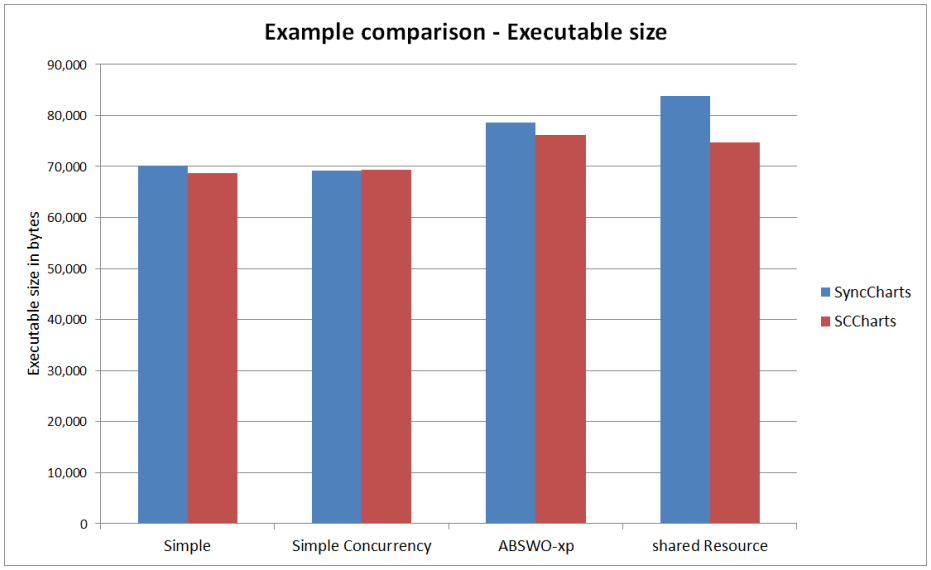


Figure 6.9. Common Example Comparison – Executable Size

6.3 Guard Evaluation

In all tests a direct connection between the SCL execution time and the count of guards in the tick function can be observed. To further examine this observation Figure 6.10 illustrates the timing results of the SCL simulations in connection with their guard counts.

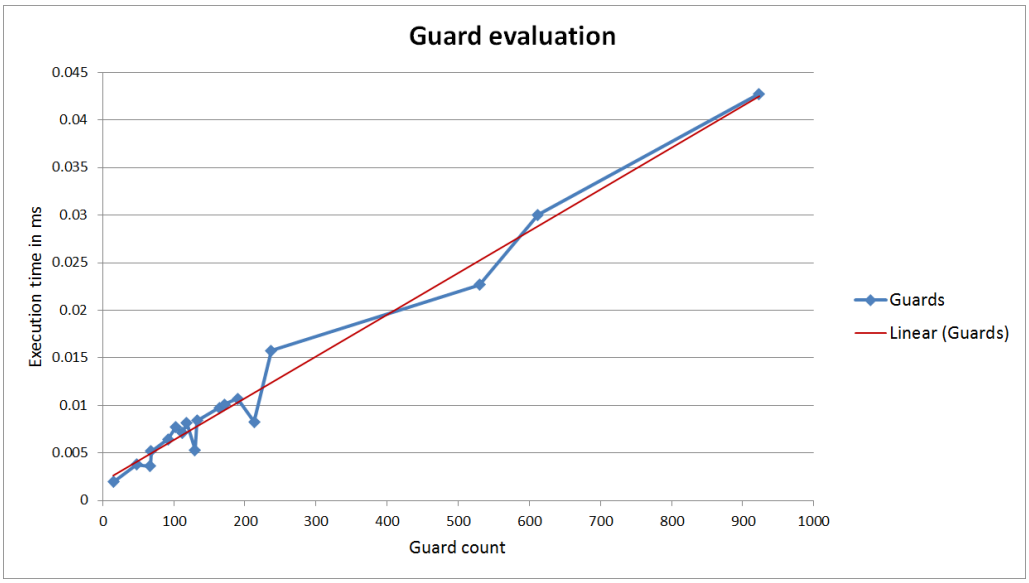


Figure 6.10. Guard Evaluation

As depicted, the execution times of all SCL programs are relatively linear depending on their guard count regardless of what kind of structure the original SCChart contained. The input data had also little impact on the timings.

This is no big surprise since the evaluation of one guard translates to one variable assignment and in the case of an output emission to a conditional. Only a synchronizer transforms to a more complex construct and therefore needs more execution time than a single guard.

Chapter 7

Conclusion

This final chapter summarizes the particular steps of the code generation approach presented in this thesis and its evaluation. It closes with ideas for future work.

7.1 Summary

This thesis described in detail a chain of transformations necessary to compile a generic sequential tick function originating from SCCharts. This tick function is meant to be an initial point for further software or hardware syntheses.

At first, the mandatory language concepts of the approach are exemplified with emphasis on sequential constructiveness. This includes the graphical modelling language SCCharts, as well as the newly derived DSL SCL and its graphical representation SCG. SCL was then further refined to an sequential program stripped of any pause registers and concurrency. The part closes with an introduction of S, the abstract language implemented in KIELER, which was used to compare the approach with SyncCharts in the evaluation.

Subsequently, the thesis elucidates each transformation step of the approach and the analyses necessary to accomplish these. It comprises transformation from SCCharts to SCL, the synthesis of the SCG and from SCL to its sequential variant. To fulfil this task, it was explained how the underlying metamodels are examined and information about data dependencies and Basic Blocks was gathered.

The evaluation showed that SCL is able to compete with the already present SyncCharts approach while incorporating the Sequentially Constructive Model of Computation and therefore accepting a wider class of programs.

7.2 Future work

Comparatively speaking, the SCL is in the early stages of development. Although the concept was largely refined over the last year, this first practical presentation may only be the beginning. A few ideas for future progress are given here.

SCG-Normalized Core SCCharts

The Normalized Core SCCharts approach, mentioned in Section 4.1.7, appears to be very promising. An SCChart translated to a normalized SCChart can directly be visualized in its SCG form without an intermediate SCL transformation. As explained in Section 4.1.7

the NCSC only contains its elemental connector types and is therefore simple to describe although it is semantically equivalent to the other SCCharts variations. Furthermore, if the transformation is carried out transiently, the SCG corresponding to the Extended SCChart can be displayed instantaneously.

Data Dependency Analysis Improvements

The data dependency presented in this thesis is made relatively conservatively. If a cyclic data dependency is detected, the program is rejected regardless of any preceding data evaluation. For instance, the rejected program in Figure 7.1 can be scheduled since both assignments containing a crossing dependency are exclusive due to the preceding conditionals. In practice, the analysis to find these may be more complex as depicted in the example since these exclusive control flows may be nested arbitrarily.

Moreover, the dependency analysis does not take registers into account. Currently rejected programs due to dependencies in concurrent regions might be schedulable if their dependent assignments do not happen in the same tick instance. Here too, lies potential to further refine the determination and accept a broader range of programs.

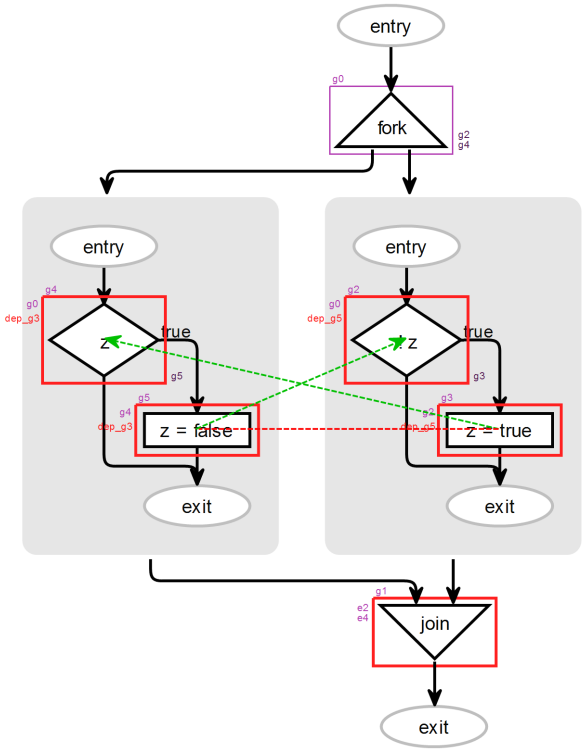


Figure 7.1. Rejected Program Example

SCG Editing

At the time of writing, the SCG visualization uses KLighD to display a textual SCL program in graph form in a *view*. There is no mechanism which permits direct editing of the SCG other than the alteration of the corresponding SCL code. A graphical editor which allows direct adding, modification or deleting of SCG nodes could be useful. This would further facilitate the creation of a transformation from the control flow graph to a textual and ultimately graphical model.

Code Generation Improvements

As optimizations the thesis only offers the most basic ideas. More matured code inquiries may improve the generation of SCL. States may be better arranged to facilitate present optimizations and similar code parts may be merged.

As the evaluation has shown, the execution time of the compiled sequential SCL program directly depends on the count of guards in the tick function. Therefore, the efficiency of the program may increase if guards were comprised less conservatively or combined to greater units.

Dedicated Simulation Engine

Even though S performs quite well in its task to serve as an intermediate language to simulate SCL and subsequently compile to Synchronous C, it is not the best fit to simulate SCL since it is meant as signal driven abstract language with a statechart-like structure. Own simulation and back-end code generation components would probably increase the efficiency of SCL.

Acknowledgements

In conclusion I would like to express my appreciation to the following persons who have accompanied me on my scholastic journey.

Prof. Dr. Reinhard von Hanxleden who continually committed himself without let-up to the students of the Department of Computer Science. Please accept my sincere thanks for not only making it possible for me to write my diploma thesis at the Chair of the Real-Time and Embedded Systems Group but also for the trust you have put in me by employing me as an assistant over the last two years. I have learnt a great amount and met many amazing people during this time.

My advisor *Dipl.-Inf. Christian Motika* for countless hours of consideration, mutual teaching and of course fun. Your engagement and professionalism will always be shining examples for me. Thank you very much for the wonderful experience.

Dipl.-Inf. Christian Schneider for your help in the context with Xtend and Xtext while working on my thesis. Thank you for putting a number of my worries to rest.

Dipl.-Inf. Christoph-Daniel Schulze for your assistance with general layout questions and KLay layered. Thank you for your speedy support and the amusing hours by my side.

Thanks go to *Dipl.-Pharm. Kirsten Petersen* for cross-reading my thesis.

Thanks go to my fellow students *Dipl.-Inf. Mathias Lichtner*, *Dipl.-Inf. Hendrik Schnepel* and *Dipl.-Inf. Jan Schaumlöffel*. My journey would not have been the brilliant experience it has been without you.

Last but not least I would like to thank my family, my brother *Robert S. Smyth jun.*, my parents *Robert S. Smyth sen.* and *Waltraud F. M. Smyth* who made my studies possible in the first place.

Bibliography

[Ame10] Torsten Amende. Synthese von SC-Code aus SyncCharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf.

[And96] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[AT00] Jauhar Ali and Jiro Tanaka. Converting Statecharts into Java code. In *Proceedings of the Fourth World Conference on Integrated Design and Process Technology (IDPT '99)*, Dallas, Texas, June 2000. Society for Design and Process Science (SDPS).

[BC84] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *LNCS*, pages 389–448. Springer-Verlag, 1984.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

[Ber99] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999. ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps.

[Ber00] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000. ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf.

[CHP06] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT '06)*, Seoul, South Korea, October 2006.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with State Machines. In *ACM International*

Conference on Embedded Software (EMSOFT'05), Jersey City, NJ, USA, September 2005.

[Dud12] Björn Duderstadt. Scccharts: A sequentially constructive statecharts dialect. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2012.

[Edw03] Stephen A. Edwards. *Tutorial: Compiling Concurrent Languages for Sequential Processors*, 2003. URL: http://www.cs.columbia.edu/~sedwards/papers/edwards2003compiling.pdf.

[Gre12] Tim Grebien. Managing academic eclipse-based projects. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2012.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Har13] Wahbi Haribi. A synccharts editor based on yakindu sct. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2013.

[Lee06] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[Mat10] Michael Matzen. A generic framework for structure-based editing of graphical models in Eclipse. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mim-dt.pdf.

[MFvH10] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010*, GI-Edition – Lecture Notes in Informatics (LNI), pages 891–896, Leipzig, Germany, September 2010. Bonner Köllen Verlag.

[Mot09] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf.

[Mül10] Martin Müller. View management for graphical models. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mmu-mt.pdf.

[MvHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. Programming deterministice reactive systems with Synchronous Java (invited paper). In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*, IEEE Proceedings, Paderborn, Germany, June17/18 2013.

[Sch11] Christoph Daniel Schulze. Optimizing automatic layout for data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2011.

[Spö09] Miro Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf`.

[SSvH12a] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. Transient view generation in Eclipse. In *Proceedings of the First Workshop on Academics Modeling with Eclipse*, Kgs. Lyngby, Denmark, July 2012.

[SSvH12b] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. Transient view generation in Eclipse. Technical Report 1206, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, June 2012. ISSN 2192-6247.

[TAvH11] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. Compiling SyncCharts to Synchronous C. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*, pages 563–566, Grenoble, France, March 2011. IEEE.

[vHMA+13a] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. SCCharts—Sequentially constructiove statecharts for safety-critical applications. 2013.

[vHMA+13b] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'13)*, Grenoble, France, March 2013. IEEE.

[Was03] Andrzej Wasowski. On efficient program synthesis from Statecharts. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compilers, and Tools for Embedded Systems (LCTES'03)*, volume 38, issue 7, June 2003. ACM SIGPLAN Notices.

[Was04] Andrzej Wasowski. Flattening statecharts without explosions. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 257–266, New York, NY, USA, 2004. ACM Press.

Index