Interactive Incremental Hardware Synthesis

for SCCharts

Francesca Rybicki

Bachelor Thesis
March 2016

Kiel University
Prof. Dr. von Hanxleden
Real-Time and Embedded Systems

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

SCCharts [vHDM$^+$14] is a synchronous visual language used for the modeling of embedded reactive systems. SCCharts are under development within the context of KIELER. KIELER is a research project of the real-time and embedded systems group of Kiel University. The KIELER Compiler (KiCo) provides various model-to-model transformations originating in SCCharts and uses the interactive incremental approach [MSvH14]. Models transformed by the KIELER Compiler within its compilation chain have the same behavior as the original SCChart model but come with various information content.

Adapting the incremental approach, this thesis proposes a further transformation. It synthesizes circuits from SCCharts by using a sequentialized representation of SCCharts to identify data-flow dependencies. Those dependencies occur if multiple writes to one variable exist within one program. In a first step the sequentialized model is transformed into a Static Single Assignment [CFR$^+$91] form, resolving the data-flow problems. This form then is synthesized into hardware which can be visualized and simulated in the KIELER SCCharts framework. Integrating the transformations listed above for the hardware synthesis guarantees the tool chain and the model-to-model transformations to be continuous and incremental. Since the KIELER SCCharts framework offers a multifunctional user interface, the user may interact with the modeled program. More precisely, the user may choose target models and modify the original model as well as simulate models individually or side by side. This interactive incremental improves understandability, traceability or optimization issues.

Contents

List of Figures

List of Tables

Abbreviations

ALU	Arithmetic Logic Unit
BB	Basic Block
CFG	Control Flow Graph
CPU	Central Processing Unit
EMF	Eclipse Modeling Framework
FPGA	Field Programmable Gate Array
FSM	Finite-State-Machines
IDE	Integrated Development Environment
ISE	Integrated Software Environment
iur	*initialize-update-read*
HDL	Hardware Description Language
KAOM	KIELER Actor Oriented Modeling
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KiCo	KIELER Compiler
KIEM	KIELER Execution Manager
KLighD	KIELER Lightweight Diagrams
MDE	Model-Driven Engineering
MoC	Model of Computation
MUX	Multiplexer
MV-SIS	Multi-Value Logic Synthesis and Verification
RCP	Rich Client Platform
SCG	Sequentially Constructive Graph

SCL	Sequentially Constructive Language
SLIC	Single-Pass Language-Driven Incremental Compilation
SC MoC	Sequentially Constructive Model of Computation
SCPDG	Sequentially Constructive Program Dependency Graph
SCT	SCCharts Textual Language
SMoC	Synchronous Model of Computation
SSA	Static Single Assignment
VHDL	Very High Speed Integrated Circuit Description Language

Chapter 1

Introduction

Meanwhile, synchronous reactive systems are omnipresent. Such systems run at the same speed as their environment they are embedded in. They interact with their environment in three steps: 1) In the first step they receive input from the environment, 2) followed by the computation of the reaction on the inputs. 3) Finally, these computed outputs are sent back to the environment. Since such reactive systems are often used in automotive industry, aerospace industry or in medical devices a deterministic behavior is essential. Programming languages such as C or Java providing concurrency do not meet the degree of determinism required for synchronous executions. Therefore, appropriate synchronous languages have been developed.

1.1 Synchronous Languages

With deterministic behavior as an essential precondition for reactive systems, programming languages such as C or Java are not sufficient for the description of such systems. Concurrent running threads and resulting race conditions do not effect determinism. For that reason synchronous languages such as Lustre [HCRP91], Esterel [BG92] or SynchCharts [And96] have been designed. The reaction chain described above, caused by an input in reactive systems, is called macro step or *tick*. *Ticks* are considered atomic. The Synchronous Model of Computation (SMoC) subdivides time into disrcete ticks. The computation during one tick takes place in multiple micro steps. According to the synchrony hypothesis[PBEB07a] a system runs perfectly synchronous if no time is consumed for the computation. That means outputs are emitted to the environment as soon as inputs from the environment are read. Hence, a tick is executed in zero time. To ensure determinism, the synchronous languages prevent multiple assignments to one variable within one and the same tick.

1.1.1 Sequential Constructiveness

Introduced by von Hanxleden et al. in 2013 [vHMA+13], the Sequentially Constructive Model of Computation (SC MoC) aims to guarantee determinism while not being overly restrictive. Therefore, the *initialize-update-read* (*iur*) protocol is introduced: The *iur* protocol allows multiple reads from and writes to the same variable within one and the same tick and still ensures determinism. This requires the usage of variables instead of signals which can either be absent or present within one tick.

1.1.2 Sequentially Constructive Charts

Sequentially Constructive Charts (SCCHarts) is a visual synchronous language introduced by von Hanxleden et al. in 2014 [vHDM$^+$14]. It is specially developed and designed for synchronous reactive systems. SyncCharts is a synchronous language introduced by Charles André in 1996 [And96]. It is a graphical representation of Esterel and is the predecessor of SCCharts. While SyncCharts is restricted by the SMoC, SCCharts leverage the SMoC. SCCharts uses a from *Statecharts* [Har87] adapted notation and provides determinate concurrency. There are two form of SCCharts: *Core* SCCharts and *Extended* SCCharts. *Core* SCCharts consist of only a limited number of elementary feature. *Extended* SCCharts provide additional means to maintain expressiveness for more complex models, e. g., signals are reintroduced as *syntactical sugar*. Every *Extended* SCCharts model can be expressed with *Core* SCCharts functionality. Hence, both forms have sufficient means to express all features of the SC MoC.

1.2 The Sequentially Constructive Graph

Sequentially Constructive Graphs (SCGs) are control-flow representations of SCCharts. They allow to analyze the behavior of a modeled system, e. g., dependencies within SCCharts. Figure 1.1 shows the compile chain of the KIELER Compiler. The inplace transformations of SCGs culminate in the sequentialized SCG. This sequentialized SCG and its transformation from core SCCharts has been studied by Smyth [Smy13]. Combining all informations of previous transformation steps, the sequentialized SCG provides the basis for the circuit transformation.

1.3 Interactive Incremental Compilation

SCCharts are developed in context of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project. The KIELER Compiler (KiCo) is based on model-to-model transformations. Figure 1.1 shows the whole KIELER compile chain including feature transformations implemented in this thesis, namely *SSA SCG* and *Circuit*. In the visualization of the KiCo compile chain nodes represent

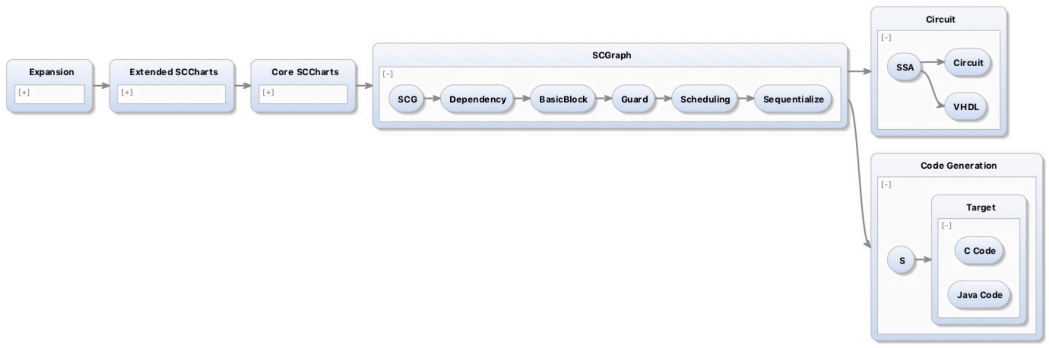


Figure 1.1. KiCo compile chain with expanded feature groups *Circuit*, SCG and *Code Generation*

all provided features. As shown, each feature transformation depends on former feature transformations. Thus, all features represent the same program while the information content differs from feature to feature. This incremental approach causes the immediate change of all feature models if the original program is modified. Thus, the KIELER application reacts to the user's inputs and performs all transformations selected by the user. This kind of interaction supports, e. g., fast detection of potential malfunction of programs and is further discussed by Motika, Smyth and von Hanxleden [MSvH14].

1.4 Hardware Synthesis from SCCharts

The challenge of maintaining determinism while supplying concurrency in programming of embedded reactive systems has been approached by several languages such as SyncCharts or Esterel. Both of these come with restrictions. SCCharts uses the SC MoC to enhance modeling of reactive systems in a less restrictive way while still guaranteeing determinism. Pursuing this approach the synthesis into hardware will be introduced in the following.

In 2013 Johannsen [Joh13] introduced a hardware synthesis from SCCharts. Johannsen chose an approach using the Very High Speed Integrated Circuit Description Language (VHDL) as target of his synthesis. The transformed VHDL programs were simulated and visualized with help of the Integrated Software Environment (ISE) tool which translates VHDL programs into circuits. Since Field Programmable Gate Arrays (FPGAs) can be programed with VHDL code the transformed program could be used on FPGAs. However, the outcome of Johannsen's transformation is only visible in an external ISE tool. Thus, modifications done in the original SCChart are not immediately visible while modeling. For each modification the whole process has to be executed from transforming the modified SCChart into VHDL code and loading this code into ISE to simulate and visualize the circuit. Hence, Johannsen's work does not cover analysis or optimization purposes nor good comprehensibility of the connection between circuits and SCCharts elements due to an uncontinuous compile tool chain.

1.5 Problem Statement

The new hardware synthesis introduced in this thesis offers considerably more opportunities for analysis, optimization and understandability for the modeler and the tool smith. Since the transformation and visualization of circuits is directly integrated into the KIELER SCCharts project, the interactive and incremental synthesis approach is adopted. Thus, each modification of the SCChart is immediately applied by the incremental and interactive transformation chain culminating in those transformations selected by the user. Moreover, each incremental step is visible for the user because each incremental result is a valid and inspectable model. This highly improves the traceability of hardware synthesis. To additionally support the comprehensibility of how the circuit derives from the sequentialized SCG, a simulation which highlights active regions of SCGs and circuits in each tick needed to be implemented. Hence the simulation facilitates understanding the dynamics of the circuit under development. On top of this, the simulation helps in validating the correctness of the hardware synthesis itself and eases development and maintenance of the compiler for the tool smith.

Since all computations in circuits run in parallel and the SC MoC proposes sequential computation the circuit has to be adjusted accordingly. Thus, circuits are not directly transformed out of the SCChart but out of the associated sequentialized SCG. Before transforming the sequentialized SCGs into circuits one intermediate step is necessary to resolve data-flow dependencies. In this step the sequentialized SCG is transformed into the SSA SCG which uses Static Single Assignments (SSAs) for each new write to a variable. Moreover, after each tick the state of the SCChart is saved to provide informations for the next tick. This state has to be saved in the circuit for the same reason. Furthermore, circuits should provide means for reset and start.

1.6 Outline of this Document

The implemented interactive incremental synthesis into hardware and all intermediate steps are discussed in the following chapters of this thesis. Chapter 2 summarizes different hardware synthesis originating in, e. g., C, Java or Esterel. It is argued why the implementation of a new hardware synthesis is necessary and which concepts and ideas of former hardware synthesis are adapted. Focusing on visualization and simulation of circuits described in Hardware Description Languages, a few tools are introduced. Furthermore, the context of the term *incremental synthesis* is set.

Chapter 3 gives an overview of used technologies for the implementation of the incremental interactive hardware synthesis. This synthesis is implemented as part of the KIELER project which uses plug-ins and Eclipse. Hence, the KIELER project itself, Eclipse and its modeling framework are introduced. Furthermore, Xtend which is a Java dialect and used for programing of the transformations is presented.

Chapter 4 presents the steps for the actual hardware synthesis conceptually. Since the hardware synthesis is integrated into KIELER and originates in SCCharts, KIELER SCCharts are introduced. The interactive incremental approach which is exemplary adapted by the KIELER Compiler is explained as well as several provided model-to-model transformations and their visual representations. One of those transformation results is the sequentialized SCG. This SCG is described in detail and is used as basis for the hardware synthesis. Before the synthesis itself is introduced, data-flow dependencies need to be resolved. This intermediate step and the developed SSA SCG without data-flow dependency problems is introduced. Finally, Capter 4 presents the created meta-model for circuits and the circuit transformation and a simulation for the synthesized circuits are presented.

Chapter 5 describes the implementation of the in Chapter 4 conceptually introduced steps.

Chapter 6 presents simulation results. Those results are used for verification and further verification methods are introduced. Moreover, the scaling of circuits is inspected.

Chapter 7 summarizes the in this thesis presented solutions for the problems stated in Section 1.5 and gives suggestions for future work addressing the topic of hardware synthesis from SCCharts.

Chapter 2

Related Work

The subject of hardware synthesis is covered by various papers which mainly challenge the parallel execution in hardware. This real parallel execution in hardware differs from the concepts of concurrency in C or Java. To interconnect both concepts, hardware synthesis means mostly a translation into a Hardware Description Language (HDL). This HDL then is executed on integrated circuits like FPGAs or in tools specially designed for simulation of HDLs.

2.1 Hardware Synthesis from C and Java

Hardware synthesis from C or Java is prevalently proposed for reasons of popularity and familiarity of both languages. Since C and Java code may be generated from SCCharts, approaches for the hardware synthesis from C or Java code are of interest. De Micheli [DM99] lists difficulties of hardware synthesis from C or C++. To use C or C++ requires a synthesizable subset of the languages to be defined in most cases. Also, e. g., concurrency and communication mechanisms need to be added.

Edwards [Edw05] focuses in his paper on two fundamental challenges concerning hardware synthesis from C-like languages. 1. concurrency and 2. timing control. Edwards introduces various languages and approaches dealing with these two topics. Exemplarily, a few languages and their benefits are given:

- HardwareC [KD90] has a C-like syntax and supports timing constraints within the language.
- C2Verilog is a compiler used to transform C code to Verilog which is an HDL. It can translate pointers, recursions or dynamic memory allocation. Soderman and Panchul [SP98], e. g., used C2Verilog for implementation of system-level algorithms in *integrated circuits*.
- Cones [SMP88] synthesizes each function in combinational blocks. Conditionals are handled by its strict C subset.

The hardware synthesis from Java is also a well discussed topic. Thomson, Chouliaras and Mulvaney [TCM06] describe a synthesis of digital hardware from a subset of the Java language. Each Java object instance corresponds to a hardware module instance. The synthesized system outputs VHDL code.

Kuhn and Rosenstiel [KR00] use Java as object oriented language and the extension *JavaBeans*. Concurrency is translated into hardware by translating each thread into control-flow graphs. These control-flow graphs are transformed into VHDL processes which may be synthesized into hardware.

Although huge progress concerning this topic is observable, the hardware synthesis from C or Java comes with too many restrictions and additional steps in workflow. These are, e. g., parallelism or timing constraints. Moreover, the SC MoC used in SCCharts and the structure of SCCharts models are designed to improve modeling, description and validation of synchronous systems. This structural model design is lost when transformed into C or Java code.

2.2 Hardware Synthesis from Statecharts

The hardware synthesis from Statecharts [Har87] introduced by Drusinsky and Harel [DH89] uses statecharts as behavioral HDL. The idea is to use single machines implementing Finite-State-Machines. The interconnection of those machines in a tree implements the behavior of the statechart.

Since statecharts have no deterministic behavior this approach is not taken into consideration for the hardware synthesis from SCCharts.

2.3 Hardware Synthesis from Esterel

Esterel [Ber00, BC85] is a synchronous language tailored for the development of embedded reactive applications in hardware and software. Esterel programs can directly be translated into circuits [Ber92]. Those circuits are divided into two sections:

1. Combinational logic, which computes the outputs and new states from the inputs and current state.

2. Sequential logic, which holds registers to store the current state of the system. In each global clock tick the state is updated.

Since SCCharts is based on an other MoC, the Sequentially Constructive Model of Computation, the semantics for hardware synthesis from Esterel [PBEB07a] is not adopted for the hardware synthesis from SCCharts. However, the ideas of conceptually separated regions and the usage of registers to store the system state is adapted. Sequentially Constructive Esterel (SCEst) studied by Rathlev et al. [RSM+15] leverages the restrictive SMoC. Thus, SCEst in addition with SSAs could be considered as a new basis for hardware synthesis in future approaches.

2.4 Hardware Synthesis from SCCharts

The subject of hardware synthesis from SCCharts has been studied by Johannsen [Joh13]. His approach translates SCCharts into the Hardware Description Language VHDL. Further, the ISE tool is used for simulation and visualization of the circuit described by the VHDL program. For additional testing purposes the VHDL code is used to program an FPGA. Each of the described steps is encapsulated within its own tool. This means that the user has to perform several intermediate steps to investigate the transformed circuit. The model as well as the VHDL code is generated in the KIELER SCCharts environment. Hence, changes to the model are immediately adopted by the VHDL code. Nevertheless the user has to use the external ISE tool to observe these changes in the circuit and on the FPGA.

The interactive and incremental approach proposed in this thesis has no breaks in the tool chain. All transformations are performed and integrated in the KIELER SCCharts environment. This means all changes to the SCCharts model are immediately observable in the transformed circuit. Furthermore, traceability of the circuits behavior is supported since all intermediate transformations originating in an SCChart are visible. This improves comprehensibility, optimization, verification and analysis potential of the synthesized hardware. However, the ideas of an intermediate transformation step into SSA form and the reset logic studied by Johannsen are conceptually adopted.

2.5 Incremental Hardware Synthesis

The main idea of an incremental synthesis approach is an improvement of comfortable usability. Changes in synthesis sources cause changes in synthesis outputs. However, considering various intermediate synthesis steps instead of only one step, the incremental approach describes an interconnection between those steps building up each other. Each step comes with different output information.

Exemplarily, Ren [Ren11] uses an incremental approach for hardware discrete controller synthesis. The proposed incremental technique also applies to communicating systems. Prasad, Anirudhan and Bosshart [PAB94] use an incremental approach for updates on gate-level implementations and reoptimization processes.

Another interpretation of incremental synthesis is introduced by Brand et al. [BDKN94]. This interpretation deals with investments program designers have when implementing. For example such an investment is the expense of physical design or simply time spent to understand the implementation. Hence, in incremental synthesis an old and a new version of design exists. Brand et al. propose a method to reuse gates from old implementations and restrict synthesis to the modifies portions only.

However, this interpretation of the term incremental synthesis is not adopted in this thesis. Furthermore, in this thesis the incremental approach is understood as a purpose for incremental model-to-model transformations.

2.6 Tooling for Hardware Description Languages

Since most of the introduced hardware synthesis use Hardware Description Languages in the following several tools for visualization and simulation of HDL programs are listed.

▷ Sigasi[1] is a free and professional tool for HDLs design such as VHDL or Verilog which may be integrated into Eclipse as well as be used as standalone application.

▷ The ISE Simulator[2] is a complete free HDL simulator which supports VHDL and Verilog.

▷ Simulink[3] is a block diagram environment for simulation and *Model-Based-Design*. Some of the key features are the graphical editor which allows managing hierarchical block diagrams, or the simulation engine with scopes and data displays for viewing simulation.

The proposed interactive incremental hardware synthesis is integrated into the KIELER framework and thus uses KIELER layout and visualization. Therefore, no external tool is necessary. Nevertheless, keeping future work in mind, the simulation tools for HDL languages are of interest.

[1]http://www.sigasi.com
[2]http://www.xilinx.com/products/design-tools/isim.html
[3]http://mathworks.com/products/simulink

Chapter 3

Used Technologies

This chapter summarizes the technologies used for the implementation of the circuit transformation. The transformed circuits are integrated into the KIELER project which is a research project of the real time and embedded systems group at Kiel University. It is based on Eclipse. Therefore, Eclipse is used for the implementation of the plug-ins for the circuit transformation. A new meta-model is deaveloped by use of Eclipse Modeling Framework (EMF). This meta-model serves as abstraction from circuits. All needed transformations leading from SCCharts into the circuit meta-model have been written in Xtend and have been integrated into KIELER. Finally, a synthesis for the visualization of circuits has been implemented in KIELER Lightweight Diagrams (KLighD) which are also part of the KIELER project.

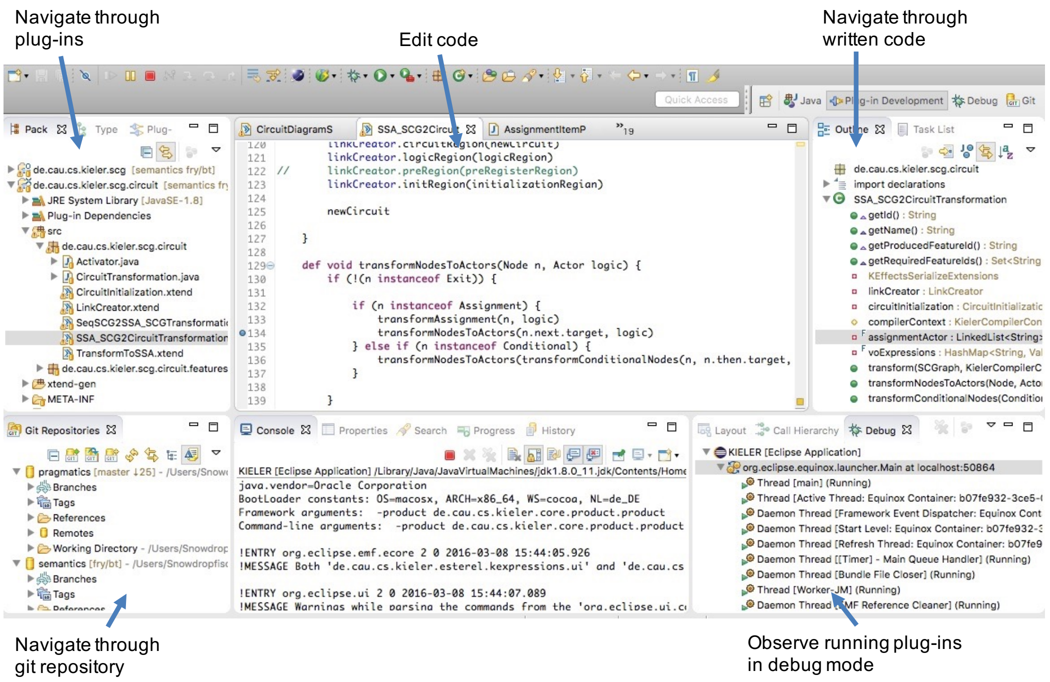
3.1 Eclipse

The plug-ins developed within this thesis have been implemented in Eclipse[1]. Eclipse is an open source Integrated Development Environment (IDE) containing a base workspace which can individually be extended by plug-ins. It is mainly written in Java and was introduced in 2001 by IBM. Although Eclipse is known for being primarily used for the development of Java applications it may also be used for development of applications in other programming languages such as C or for modeling purposes. The Eclipse Rich Client Platform (RCP) provides core components and plug-ins to extend the functionality of the RCP. This modular expandability allows incremental development of complex programs.

Eclipse provides a set of different views and editors which are put together in a workbench as shown in Figure 3.1. For example, the *Plugin Explorer* on the left side helps the user to navigate through the workspace while the *Outline* on the right side helps the user to navigate within the file opened in the editor which is located in the center.

Aside from these different views, the workbench provides many functionalities such as menu buttons, one of which is the *Run* button. Pressing this button starts

[1]https://eclipse.org


Navigate through plug-ins
Edit code
Navigate through written code
Navigate through git repository
Observe running plug-ins in debug mode

```
        linkCreator.circuitRegion(newCircuit)
        linkCreator.logicRegion(logicRegion)
//      linkCreator.preRegion(preRegisterRegion)
        linkCreator.initRegion(initializationRegian)

        newCircuit

    }

    def void transformNodesToActors(Node n, Actor logic) {
        if (!(n instanceof Exit)) {

            if (n instanceof Assignment) {
                transformAssignment(n, logic)
                transformNodesToActors(n.next.target, logic)
            } else if (n instanceof Conditional) {
                transformNodesToActors(transformConditionalNodes(n, n.then.target,
            }

        }
```

```
KIELER [Eclipse Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_11.jdk/Contents/Home
java.vendor=Oracle Corporation
BootLoader constants: OS=macosx, ARCH=x86_64, WS=cocoa, NL=de_DE
Framework arguments:  -product de.cau.cs.kieler.core.product.product
Command-line arguments:  -product de.cau.cs.kieler.core.product.product

!ENTRY org.eclipse.emf.ecore 2 0 2016-03-08 15:44:05.926
!MESSAGE Both 'de.cau.cs.kieler.esterel.kexpressions.ui' and 'de.cau.cs

!ENTRY org.eclipse.ui 2 0 2016-03-08 15:44:07.089
!MESSAGE Warnings while parsing the commands from the 'org.eclipse.ui.c
```

Figure 3.1. Editor and different views in Eclipse workbench

the plug-ins mentioned in the *Run Configuration*. This configuration specifies which plug-ins will be used in the next run of the implemented program.

3.1.1 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF)[2] EMF is an Eclipse plug-in for model-driven development and is required for the creation of meta models in this approach. EMF produces a set of Java classes for a model specification described in XMI. It consists of three fundamental pieces:

1. The core EMF framework uses a meta model (Ecore) for the description of models and provides runtime support for the models.
2. The EMF.Edit framework uses generic reusable classes for building editors for EMF models.
3. The EMF.Codegen is capable of generating everything needed to build a complete editor for an EMF model

[2]http://www.eclipse.org/modeling/emf

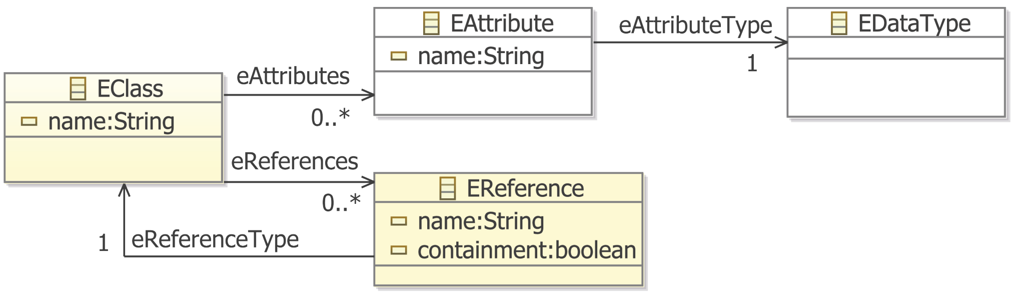


Figure 3.2. Subset of EMF meta-model [Mot09]

Modeling in EMF

Eclipse Modeling Framework (EMF) uses meta-models which are simply models which describe a set of models. The model used to represent models in EMF is called *Ecore*. Since Ecore is an EMF model it is a meta-model itself. Figure 3.2 shows a simplified subset of the Ecore meta-model depicting the classes which were used for modeling in the context of this thesis. EMF models consist of classes which may have attributes and references to other classes. While attributes must have a name and a *dataType*, references represent associations in-between classes and have a name and a boolean flag to indicate if the target class is a containment class or the a container class.

3.1.2 Xtend

Xtend[3] is a flexible and expressive dialect of Java. It integrates with EMF and is used for model-to-model transformations in the context of KIELER. Xtend is a statically-Xtend provides several improvements compared to Java source code such as *extension methods*, *lambda expressions* or *type inference*. Extension methods are where the name Xtend originates from. They allow to add new methods to existing types without modification. Lambda expressions are kind of anonymous classes with one single method. They may declare parameters without declaring types due to type inference.

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

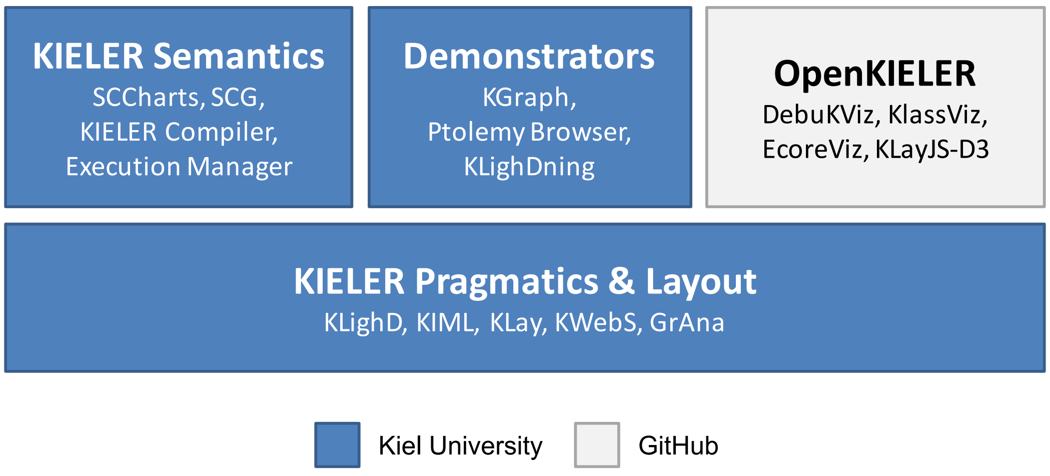

KIELER Semantics
SCCharts, SCG, KIELER Compiler, Execution Manager
Demonstrators
KGraph, Ptolemy Browser, KLighDning
OpenKIELER
DebuKViz, KlassViz, EcoreViz, KLayJS-D3
KIELER Pragmatics & Layout
KLighD, KIML, KLay, KWebS, GrAna
Kiel University
GitHub

Figure 3.3. The different areas of the KIELER project

3.2 The Kiel Integrated Environment for Layout Eclipse Rich Client

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[4] is an open source research project promoted by the Real-Time and Embedded Systems Group at Kiel University. It integrates into the Eclipse rich client platform and focuses on the enhancement of graphical model-based design of complex systems. Therefore, the idea is to consistently employ automatic layout in all components of a diagram. The project is divided into different areas depicted in Figure 3.3. In the following, a set of the areas and a part of their projects will be introduced.

3.2.1 Semantics

KIELER *semantics* consists an infrastructure for simulations of graphical modeling languages with emphasis on synchronous languages. One main contribution of this area are SCCharts as mentioned in Section 1.1.2. The approach introduced in this theses benefits from the provided infrastructure. It expands the existing simulation and incremental compilation possibilities by a hardware synthesis and a simulation of the synthesized circuits.

[4]http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER

The KIELER Compiler

The KIELER Compiler (KiCo)[5] project is used as generic framework. This framework allows to register step-by-step compilation transformations of EObjects. The transformations can be written in Xtend or Java and reach from SCCharts up to the SCG and C Code generation. The circuit transformation is a new transformation step located as *low-level* synthesis and new back end behind the *high-level* synthesis of SCGs. The KIELER Compiler is a generic framework based in the interactive incremental Single-Pass Language-Driven Incremental Compilation approach [MSvH14].

The KIELER Execution Manager

The KIELER Execution Manager (KIEM)[6] [Mot09] can be used to execute arbitrary domain-specific models. KIEM itself does not do any simulation computation but allows to easily integrate arbitrary simulation components. Simulation results can also be visualized by dedicated visualization components. The behavior of the synthesized circuit is simulated and visualized with the help of KIEM.

3.2.2 Pragmatics

In the context of MDE, the *pragmatics* area of KIELER focuses on the simplification of the daily work of modelers. Therefore, the creation and modification of models as well as the synthesis of various views on those models are topics. One project addressing these topics is KLighD. KLighD is used for the synthesis of graphical representations of models. Visualization and simulation of the in this approach synthesized circuits are realized by means of KLighD.

KIELER Lightweight Diagrams

KIELER Lightweight Diagrams (KLighD)[7] project aims to offer transient lightweight representations of models. Graphical or textual representations of components in models are created by synthesizing them into underlaying KGraph meta-model components which can be depicted by KLighD.

[5]http://rtsys.informatik.uni-kiel.de/confluence/x/aYCQ
[6]http://rtsys.informatik.uni-kiel.de/confluence/x/nwEF
[7]http://rtsys.informatik.uni-kiel.de/confluence/x/swEF

Chapter 4

Interactive Incremental Hardware Synthesis

This chapter conceptually summarizes the interactive and incremental hardware synthesis from SCCharts. The main subject, namely the transformation of SCGs into circuits (5) is described in Section 4.2.3. As all transformations are integrated into the KIELER compile chain, a brief introduction of SCCharts and of its interactive incremental compilation using the currently possible transformations provided by the KiCo will be given in Section 4.1. In this context different visualizations of SCGs are introduced in Section 4.1.4. The circuit transformation will originate from the sequentialized SCG adopting and expanding the incremental compilation of KiCo.

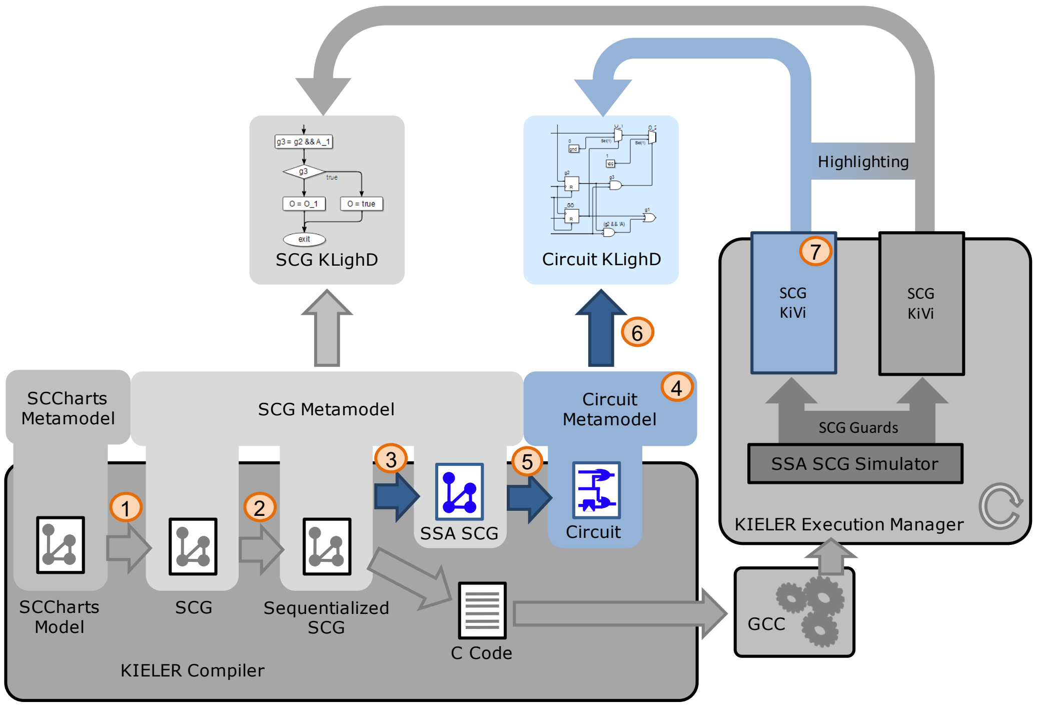


Figure 4.1. Interactive Incremental Hardware Synthesis workflow overview

Figure 4.1 presents an overview of all necessary steps from 1 to 7 to realize the transformation from SCCharts into circuits. The blue marked incremental synthesis steps are those studied and implemented as part of this thesis, namely steps 3 to 7. The overview shows how the hardware synthesis steps are integrated into KIELER and how they depend on each other.

As shown, a meta-model for circuits was created (4). Subsection 4.2.2 gives a closer look at this meta-model which serves as an abstraction of circuits. Based on this meta-model, circuits are transformed and visualized as described in Section 4.2.3.

The transformation from sequentialized SCGs into circuits needs one intermediate step which is introduced in Section 4.2.3 (3). This step is a transformation which uses Static Single Assignments (SSAs) to resolve data-flow dependencies appearing in SCGs. Eventually, the actual transformation into circuits takes place (5).

The visualization of the circuits is another model-to-model transformation and is described in Section 4.2.3 (6).

To support the understanding of the dynamic behavior of circuits depending on the corresponding SCG's behavior a simulation is proposed which also has been implemented and is introduced in Section 4.2.5 (7).

4.1 Preliminaries

This chapter introduces language concepts and their impact on the incremental hardware synthesis. Therefore, the incremental interactive approach of KiCo is presented, as well as the integration of the circuit synthesis into this approach.

4.1.1 SCCharts

Synchronous Constructive Charts is a synchronous language introduced by von Hanxleden et al. [vHDM$^+$14] designed for safety-critical reactive systems. Its visual syntax uses a *Statecharts* [Har87] notation borrowed from SyncCharts [And96]. The SCCharts semantics is based on the SC MoC which follows a synchronous approach providing determinism while being less restrictive than SyncCharts. The execution of SCCharts uses a discrete tick function as abstraction from time. In each tick, outputs are computed as reaction of inputs given to the model. There are two sets of features which can be used when modeling SCCharts.

1. Core SCCharts have all basic instructions for modeling state machines and additionally fork/join concurrency.

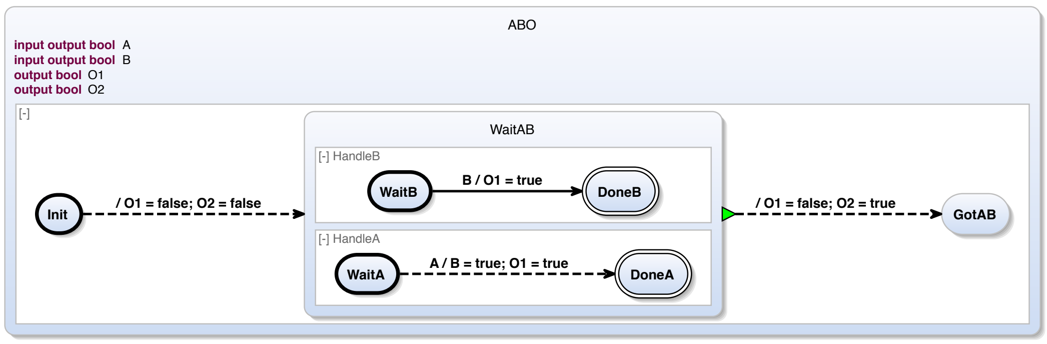


Figure 4.2. SCChart for ABO – the "Hello World!" of SCCharts illustrates concepts of *core* SCCharts

2. Extended SCCharts extend the instruction set of *Core SCCharts* by *syntactical sugar* which allows to improve readability by hiding complexity of the model. For example they provide different types of aborts or transitions such as strong aborts, weak aborts or conditional terminations. However, all these advanced instructions can be expressed in and transformed into *Core SCCharts*.

ABO Example Figure 4.2 shows ABO, the "Hello World!" of SCCharts. ABO has two boolean inputs A, B and two boolean outputs O1 and O2 as shown in the declaration interface. The initial state Init is entered when the program starts execution. In the same initial tick, the immediate transition to the state WaitAB is taken and the outputs O and O2 are set to false. The system is now in the initial states of the regions HandleA and HandleB namely WaitA and WaitB. Those regions run concurrently. Now two different execution traces depicted in Figure 4.3 are exemplarily given.

First trace: As the system is in the initial tick, the immediate transition in HandleA triggers if A is set to true in this tick. Assuming this is the case and the transition to the final state DoneA is taken, B and O1 are set to true. Since this is the first tick, the non-immediate transition in HandleB is not triggered even if B is set to true. In the second tick no input variables are set to true which means the program remains in the states WaitB and DoneA. As soon as B is set to true the transition to DoneB triggers and O1 is set to true. In the same tick but sequentially afterwards, the termination transition changes O1 to false and O2 to true.

Second trace: Here the system is not in the initial tick but in any other tick of the execution. The system is in the states WaitA and WaitB. If now A is set to true and the effect of the transition to DoneA sets O1 and B to true, which triggers the transition to DoneB, the program terminates in this tick setting O1 back to false and O2 to true.

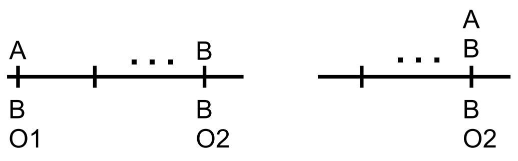


Figure 4.3. Two ABO traces

ABO illustrates concurrency and sequential overwriting of variables which is allowed according to the synchronous constructive execution semantics of SCCharts as long as a determinate scheduling exists. The sequential access to B as described above has to be guaranteed since HandleA and HandleB run concurrently and B could be read from before set to true as an effect of the transition in HandleA.

4.1.2 Sequential Constructiveness

The ABO example shows that concurrent reads and writes to shared variables and consequently possible race conditions lead to problems. Those problems are handled deterministically by the SC MoC. The SC MoC proposes a protocol which controls concurrent accesses to variables, the *initialize-update-read* protocol. This protocol is introduced by von Hanxleden et al. [vHMA+14].

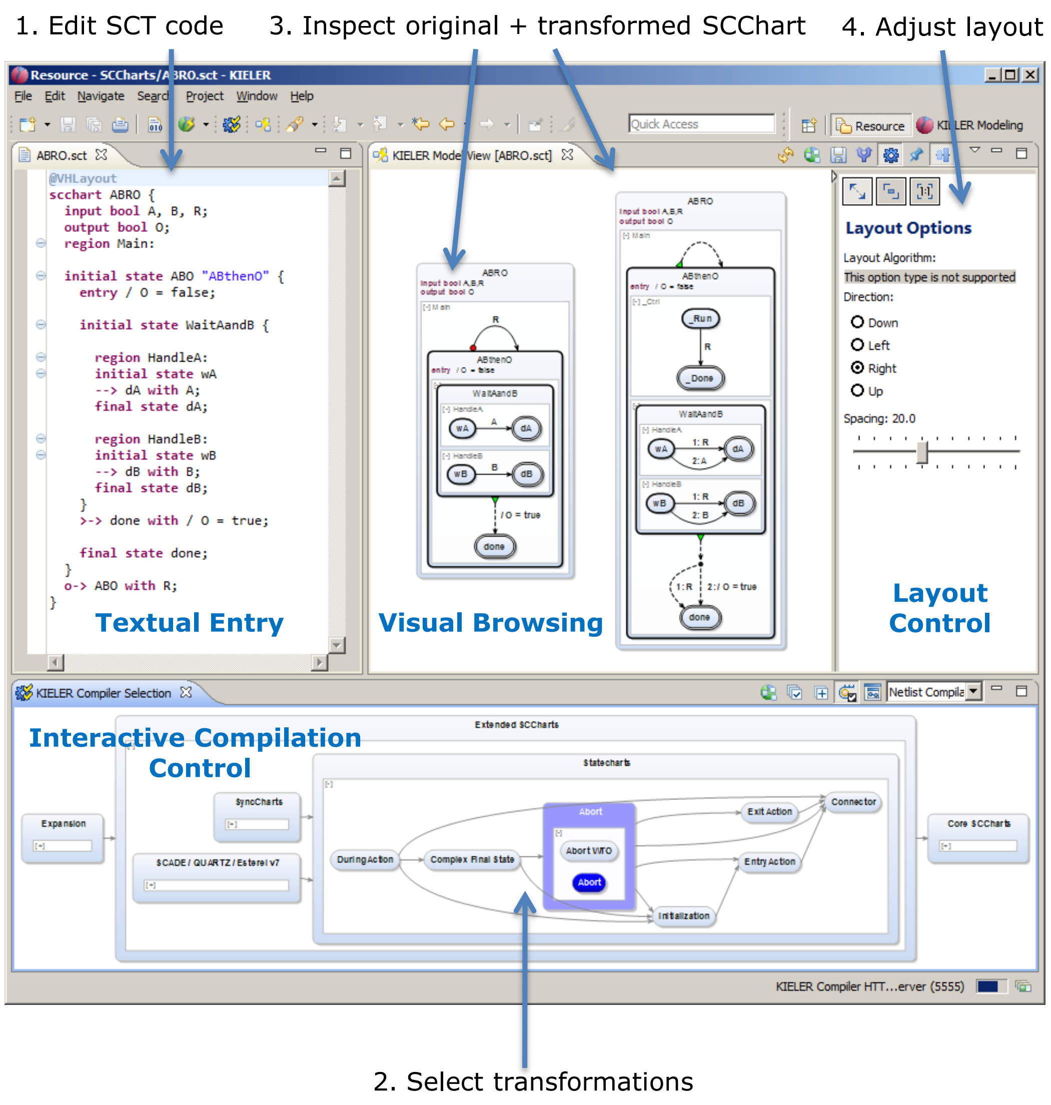
4.1.3 Interactive Incremental Compilation of SCCharts

The *interactive incremental compilation* [MSvH14] offers enhanced opportunities regarding the control over the compilation by choosing the compilation strategy and observing intermediate results. Figure 4.4 shows the interactive incremental based user story for the KIELER SCCharts tool which exhibits the ideas around interactive incremental compilation. Its key is the *Interactive Compilation Control* depicting the KiCo compilation chain at the bottom of Figure 4.4.

(1) The user may create SCCharts by describing the system to be modeled in the SCCharts Textual Language (SCT). SCT is the textual format of SCCharts. The editor view for SCT is on the left side of Figure 4.4. The SCChart as graphical diagram is automatically synthesized and shown, based on the SCT program. The visualization of the SCChart is located on the left side of the middle view.

(2) Since the SCCharts tool provides the compilation chain, the user may choose from various features the one of particular interest.

(3) The transformed model is depicted on the right side of the middle view. Hereby, the user may compare the original SCChart with the visualization of the model transformation chosen by the user. Any modification of the SCT is immediately adopted from all visualized models.


1. Edit SCT code
3. Inspect original + transformed SCChart
4. Adjust layout
Resource - SCCharts/ABRO.sct - KIELER
ABRO.sct
KIELER Model View [ABRO.sct]
Textual Entry
Visual Browsing
Layout Control
KIELER Compiler Selection
Interactive Compilation Control
Extended SCCharts
Statecharts
Expansion
SyncCharts
SCADE / QUARTZ / Esterel v7
During Action
Complex Final State
Abort
Abort WTO
Exit Action
Entry Action
Connector
Initialization
Core SCCharts
KIELER Compiler HTT...erver (5555)
2. Select transformations

```
@VHLayout
scchart ABRO {
  input bool A, B, R;
  output bool O;
  region Main:

  initial state ABO "ABthenO" {
    entry / O = false;

    initial state WaitAandB {

      region HandleA:
      initial state wA
      --> dA with A;
      final state dA;

      region HandleB:
      initial state wB
      --> dB with B;
      final state dB;
    }
    >-> done with / O = true;

    final state done;
  }
  o-> ABO with R;
}
```

Layout Options

Layout Algorithm:
This option type is not supported
Direction:
Down Left [✓ Right] Up
Spacing: 20.0

Figure 4.4. Screen shot of KIELER SCCharts tool adopted from [MSvH14]. Annotations for user story for interactive model-based compilation.

(4) On the right side of Figure 4.4 layout options for the depicted models are listed. The user may choose different kinds of options to adjust the model visualization for their purposes.

Single-Pass Language-Driven Incremental Compilation

The incremental model-based compilation strategy [MSvH14] is based on a set of model-to-model transformations. These transformations originate in a source model and are consecutively constructed culminating in some target. The essential properties of SLIC are as follows.

`Single-pass` means that every transformation is only performed once.

Since each transformation step depends on the former compiled transformation it can be considered as **`increment`** of the compilation.

The language features and their transformations determine the order in which the transformations are applied. This is the **`language-driven`** approach.

All **`intermediate results`** of the transformations can be expected as standalone models which the user may inspect.

4.1.4 The Sequentially Constructive Graph

The Sequentially Constructive Graph (SCG) is used as an alternative representation for core SCCharts to ease down-stream compilation and to support understanding the incremental interactive compilation steps. The SCG is created as a model-to-model transformation from normalized *Core* SCCharts. It is a set of statement nodes and control-flow edges. The node types are *entry* and *exit*, *assignments*, *conditionals*, *forks/joins* and *surfaces/depths* which delimit tick boundaries. The edge types are *flow* edges, *dependency* edges and *pause* edges. Figure 4.5 shows how core SCChart elements are mapped to SCG components. It further shows how SCG components are translated into hardware.

The SCG comes with several different representation options and further model-to-model transformations some of which are displayed in the following. All options provide different informations. Nevertheless, it is possible to draw the SCG without these options.

SCG Dependency Representation

The SCG provides means to illustrate the different types of dependencies regarding concurrent access to shared variables. Those dependencies are:

- write - write dependencies,
- absolute write - relative write dependency,
- write - read dependency and
- realtive write - read dependency.

Those dependency representations are useful for analyzing, e. g., causality and schedulability problems.

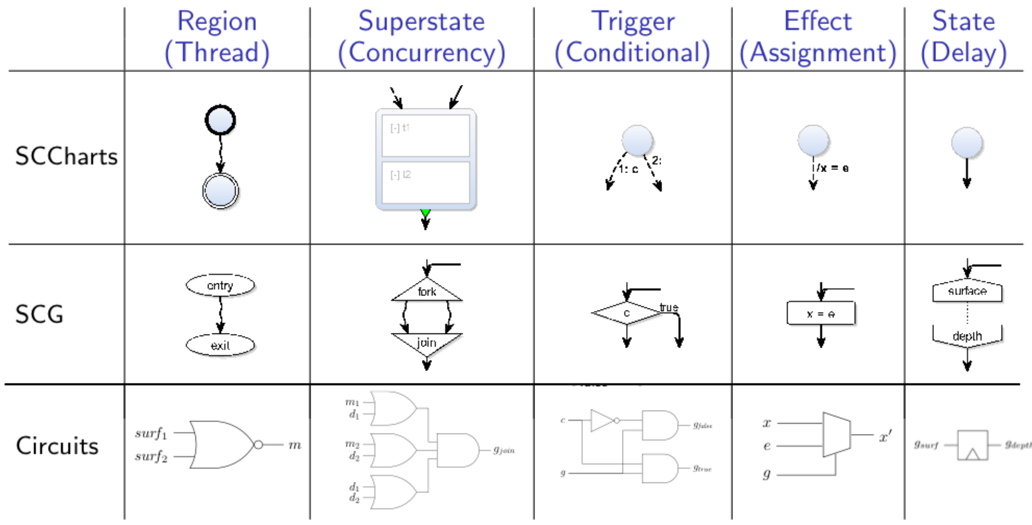


Figure 4.5. Mapping of core SCCharts elements into SCG elements and hardware. Adopted from [MSvH14]

SCG Basic Block and Guard Visualization

Basic Blocks (BBs) qualify parts of the SCG in which the control-flow does not branch or join different branches. That means a BB can be executed as single block without rescheduling. The most conservative way of splitting the SCG up into BBs is to qualify each node as BB. However, in SCGs as many nodes as possible are summarized in one BB while any node must not occur in two different BBs.

Every BB has a *guard* which determines if its BB is either active or inactive in the current tick. A BB is active if the dependencies to previous BBs or data dependencies expressed in its guard are evaluated to true. Figure 4.6 depicts the SCG for ABO wit BBs and guards.

If the result of the dependency analysis and the BB analysis as described by Smyth [Smy13] is that the program contains no cyclic dependencies the transformation into sequentialized SCGs is possible since no causality problems exist. A sequentialized SCG has no concurrency and the control-flow is executed in each tick in due consideration of the BB dependencies and the current state of the program.

All considerations regarding dependencies and scheduling have been taken into account resulting in the sequentialized SCG. For that reason the circuit transformation will be built on sequentialized SCGs. Additionally, sequentialized SCGs only consist of conditional and assignment nodes which simplifies the transformation into circuits.

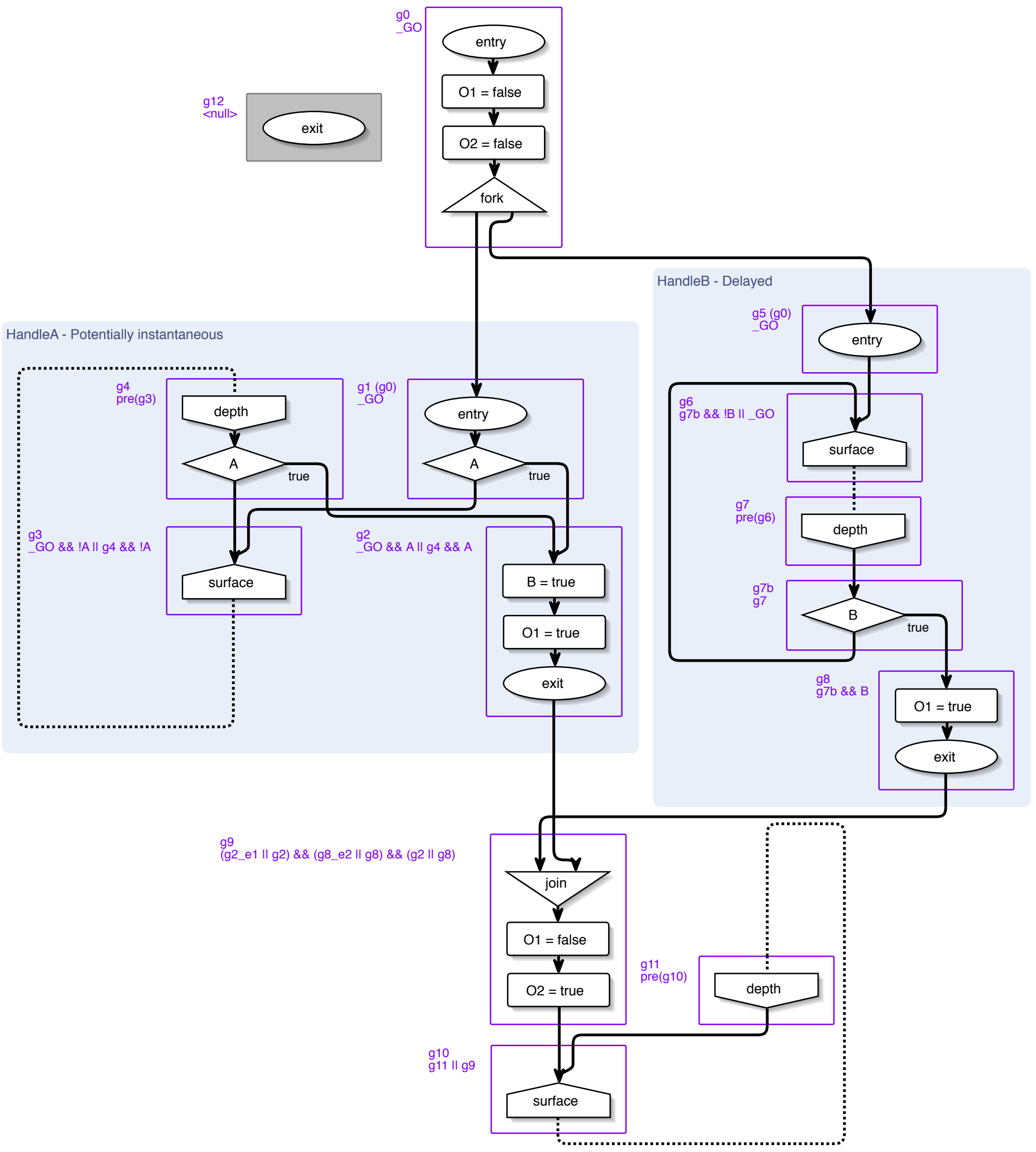


Figure 4.6. SCG for ABO with Basic Blocks and guards

Sequential SCG

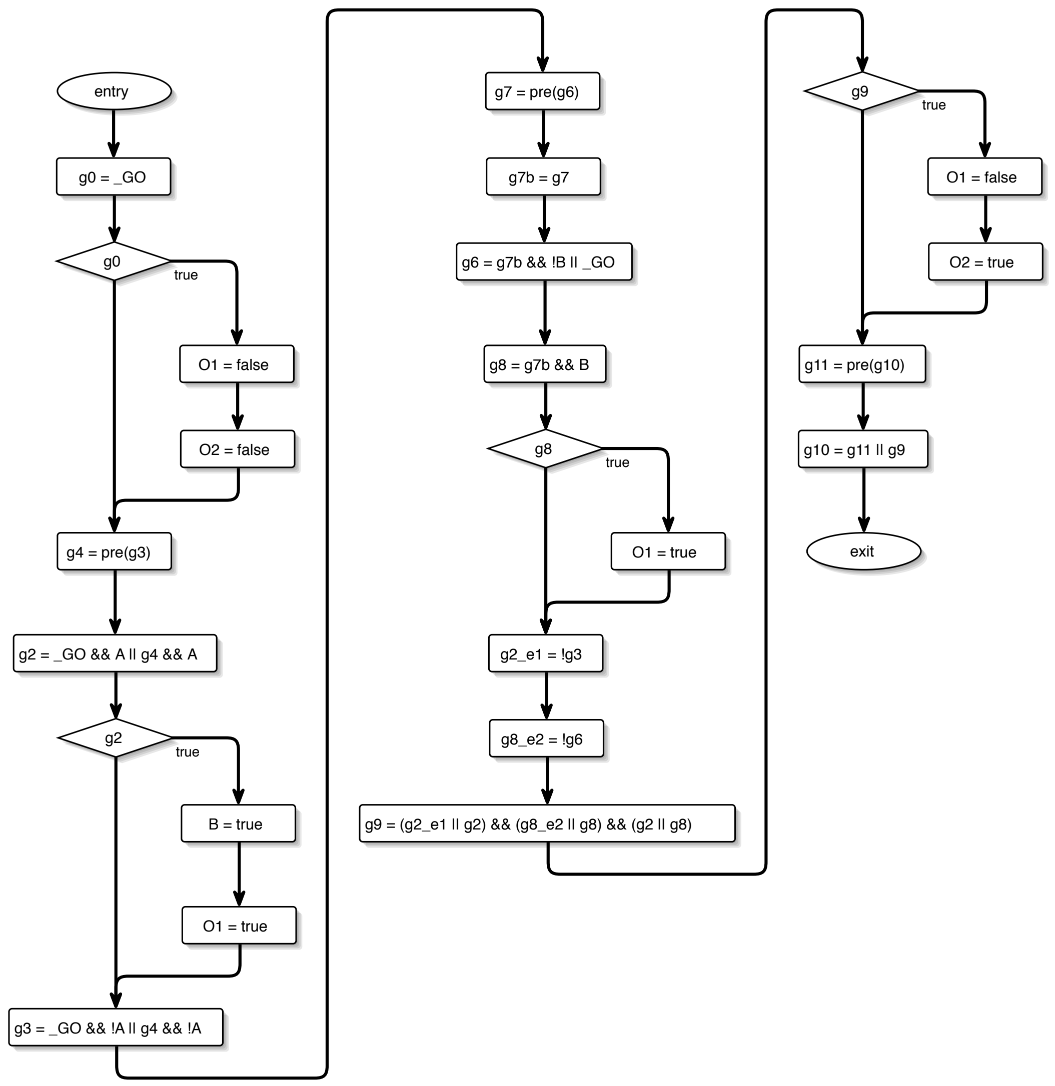
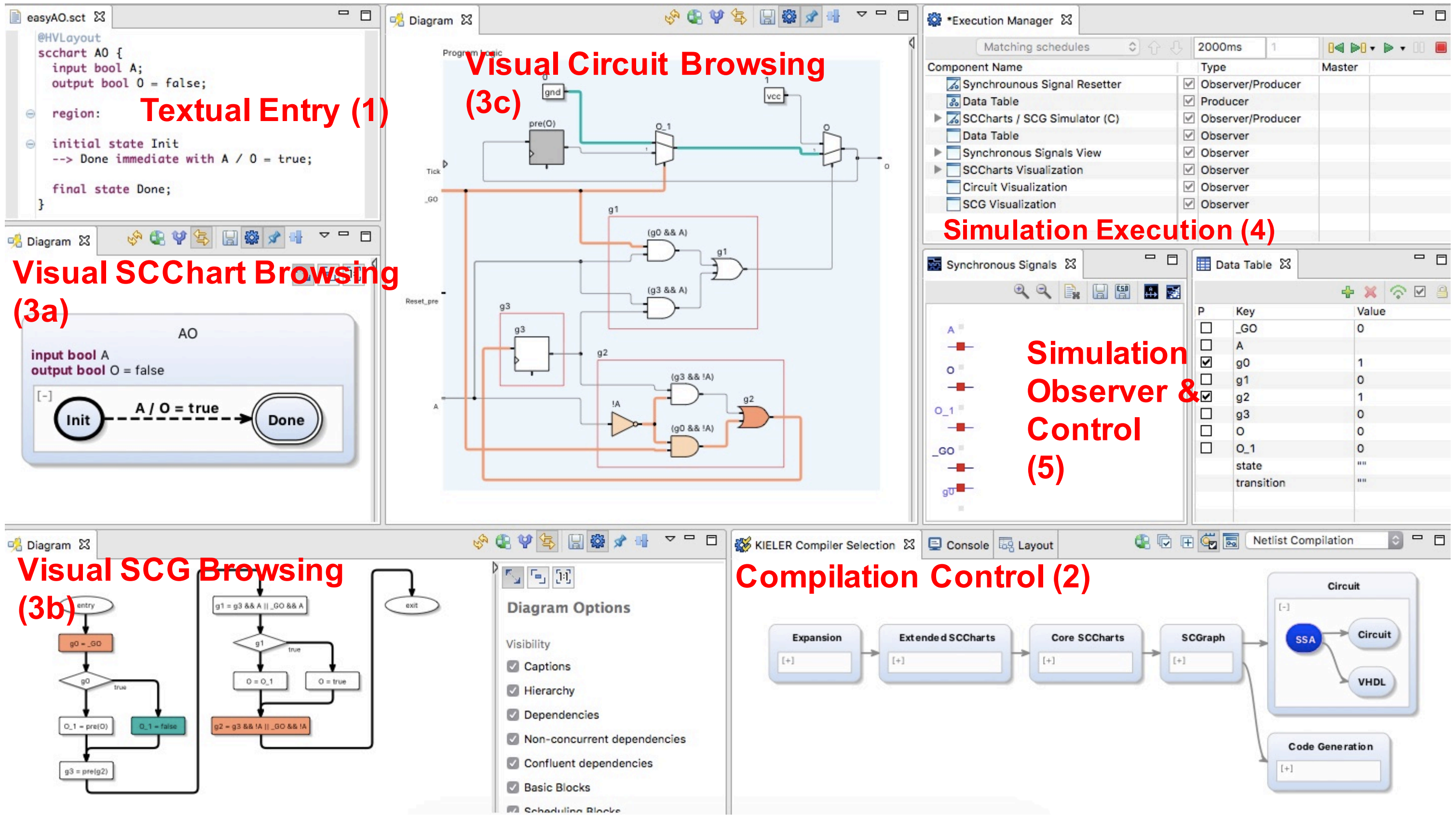


Figure 4.7. Sequential SCG for ABO

The set of assignment nodes is composed by assignment nodes carried over from the SCG and new assignment nodes representing the guards and dependency expressions of the BBs.

Each time a BB in the SCG contains an assignment, the transformation into sequentialized SCGs invokes the creation of a conditional node containing the BB's guard as expression. This conditional node is subsequently used in the sequentialized SCG. Figure 4.7 shows the sequentialized SCG for ABO.



```
@HVLayout
scchart AO {
  input bool A;
  output bool O = false;

  region:

  initial state Init
  --> Done immediate with A / O = true;

  final state Done;
}
```

Component Name	Type	Master
Synchrounous Signal Resetter	☑ Observer/Producer	
Data Table	☑ Producer	
SCCharts / SCG Simulator (C)	☑ Observer/Producer	
Data Table	☑ Observer	
Synchronous Signals View	☑ Observer	
SCCharts Visualization	☑ Observer	
Circuit Visualization	☑ Observer	
SCG Visualization	☑ Observer	

P	Key	Value
☐	_GO	0
☐	A	
☑	g0	1
☐	g1	0
☑	g2	1
☐	g3	0
☐	O	0
☐	O_1	0
	state	""
	transition	""

Figure 4.8. Screenshot of KIELER SCCharts tool annotated with high-level user story for incremental interactive model-based hardware synthesis

4.2 Hardware Synthesis

Starting with step number 4 as depicted in Figure 4.1, the hardware synthesis is conceptually explained in the following. This chapter will provide all information leading to the visualization of circuits representing the same logical behavior as the corresponding sequentialized SCG. At first, the user story of Section 4.1.3 is expanded and the new features are explained. Hereafter, the subsequent sections describe each step leading to the final result.

4.2.1 Userstory for Incremental Interactive Hardware Synthesis

The incremental interactive model-based compilation approach introduced by Motika, Smyth and von Hanxleden [MSvH14] is a strategy for the realization of the abstract compilation concepts exemplified by the KIELER SCCharts compilation. Referring to the user story given by Motika, Smyth and von Hanxleden [MSvH14] the new incremental transformation for hardware synthesis and interactive will be introduced in the following. Consider the extended user story depicted in Figure 4.8:

(1): The textual entry window serves the user as editor for the model. The textual representation of a model is written in SCT. In this case `AO` is shown. It remains in the initial state until the boolean input `A` is set to true to emit a boolean output `O` and then terminate.

(2): The interactive compilation control window allows the user to select different model-to-model transformations. Those transformations incrementally depend on each other. The new feature group **`Circuit`** contains the interactive incremental hardware synthesis introduced in this thesis.

(3): The visual browsing windows let the user see the result of the transformations selected in step (2). The depicted visualizations correspond to the SCT program (1). (3a) is the visualization of the modeled SCChart , (3b) shows a SSA SCG and (3c) shows the hardware circuit. (3b) and (3c) are the visualized results of the transformations which will be introduced in the following sections. If the user decides to modify their model written in (1), all active visual browsing windows will accordingly adjust their transformation visualization.

(4): In the simulation execution window the user may add components to simulate the program in the in (2) selected transformation. If the simulation is started, in each tick the active components of the visualized model will be highlighted. This improves the dynamic comprehensibility of the circuit.

(5): During the run of a simulation the user may set input variables and thus observe the reaction of the system for different kind of situations.

Hardware Synthesis from SCCharts

Johannsen [Joh13] studied and implemented the synthesis of VHDL code from SCCharts and visualizes circuits with the help of the ISE. The new approach here is to integrate the visualization directly into the compile chain to maintain the incremental interactive usage. An intermediate step which transforms sequentialized SCGs into SSA SCGs is implemented and also available as model-to-model transformation in the compile chain. Hence, the new approach extends Johannsen's work and gives improvements by adopting the incremental interactive compilation approach.

Table 4.1. Comparing the predecessor SCL2VHDL [Joh13] with the introduced circuit tranformation

	SCL2VHDL Project	SCG2Circuit Project
VHDL/circuits from SCCharts	+	+
VHDL/circuits from SCGs/SCL	+	+
Visualize circuits	+	+
Directly program FPGAs	+	+/-
Open source / no license needed	-	+
Seamless tool-chain	-	+
Circuits visible while modeling	-	+
Simulation visualization	-	+
Side-by-side co-simulation	-	+
Understanding circuits	-	+
Element tracing: SCChart ↔ circuit	-	+
Reuse SSA representation	-	+

Table 4.1 compares both projects. Both projects deal with the hardware synthesis from SCCharts and hence a transformation from SCCharts into circuits exist in both projects. Since SCGs and their textual representation, e. g., the Sequentially Constructive Language (SCL), are transformed from SCCharts, a circuit transformation s in both cases and the circuits may be visualized. While Johannsen uses the external ISE tool, the new approach visualizes the circuits directly in KIELER. The latter enables a seamless tool-chain and the circuit is visible while modeling. Further, the interactive incremental approach makes the circuit immediately adopt

every change of the model. The VHDL transformation in Johannsen's hardware synthesis requires an intermediate step by loading the program in ISE to visualize the circuit. Hence, changes to the original model are not immediately adopted. Furthermore, ISE allows to load the program on FPGAs. In the new approach no VHDL transformation exists at time of writing. This transformation is a desired step which ought to be implemented to tie in with Johannsen's approach. Since the new circuit transformation is integrated in KIELER, the KIELER infrastructure may be used for simulation of the circuits and co-simulation of circuits and other models. The incremental compilation helps to understand circuits by depicting each intermediate feature the circuit transformation builds up on. Particularly the SSA SCG transformation is investigatable by the user as it is an intermediate step in the KiCo compiler chain.

4.2.2 Circuit Meta-Model

Step 4 in Figure 4.1 depicts the creation of a meta-model by means of which circuits may be represented. This chapter introduces the resulting requirements as well as limitations concerning the meta-model. The resulting meta-model will strongly be based on the KAOM meta-model [MSF$^+$11].

Requirements

The overriding objective was to create a meta-model appropriate for the representation of circuits. However, with regard to future work and usage of this meta-model, its highly generic design suggested itself.

Generally speaking, circuits are composed of wires, different kinds of logical gates, which include the arithmetical unit, and memory units. Wires are used to connect those gates or memory units with each other forwarding signal informations. Moreover circuits react to input signals given by the environment. Those signals will be processed within the circuit and output signals addressing the environment are emitted.

Hence, the meta-model should provide means to depict input and output signals as well as logical gates, memory units and wires. Furthermore, means to describe which parts of the circuit are connected by wires should be provided. Furthermore, each wire should have a unique source and a unique target. This restriction serves to keep the meta-model simple to prevent hyperedge usage.

As known from, e. g., Arithmetic Logic Units (ALUs) in CPUs, it is desirable to encapsulate some parts of circuits within blocks of the same. This is done to enhance readability and for simplification. For that reason a hierarchical structure of circuit components is supported by the meta-model.

Table 4.2. Circuit meta-model requirements and corresponding components and concepts of the meta-model.

Requirement	**Meta-Model Component**
Depiction of circuit contents - Wire - Gate - Entry/Exit ports	Created - Class Link - Class Actor - Class Port
Hierarchical Structure	Nested actors, containment dependencies
Make annotations possible	Abstract class NamedObject
Linking constraints	Abstract class Linkable
Reusability	Unspecified design - only three classes - no enumerations

Since the meta-model may potentially be reused in future projects with other subjects than circuits, the meta-model should be generic and provide means to model any kind of flow diagram. Furthermore, within all parts of the meta-model annotations give the opportunity to attach additional generic information. Table 4.2 summarizes all meta-model requirements.

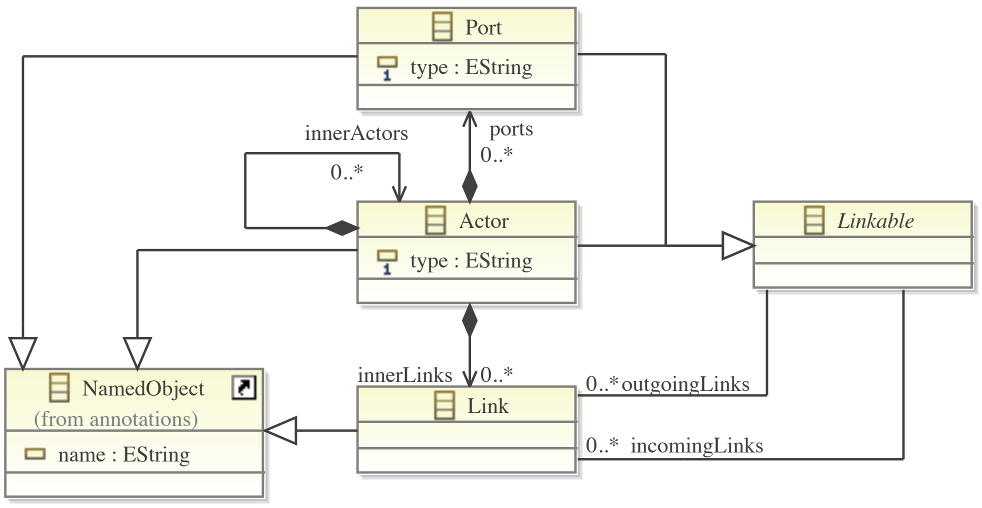


Figure 4.9. Circuit meta-model

Circuit Meta-Model

The meta-model as presented in Figure 4.9 meets all requested requirements.

The abstract class `Linkable` declares which object classes may be connected by links. In this case linkable objects are of class `Actor` and `Port`. The `Actor` class may be used for the depiction of logical gates and memory units, such as Registers or FlipFlops. The hierarchical structure is achieved through `innerActors` as possible containment of an actor object. The `Port` class objects serve as connection points between links and actors. Nevertheless, their usage is optional since actors themselves are linkable. While all linkable objects can have several incoming and outgoing links, an object of class `Link` itself has at most one source and one target linkable object.

Due to the objective of an utmost generic design, no enumerations for, e. g., gate types have been implemented. To differentiate between types of actors or ports, it is possible to assign an ID as type. Additionally, all three implemented classes inherit from the abstract class `NamedObject` to assign a name to each object. Moreover, the use of this class provides the opportunity to assign annotations to objects.

4.2.3 Circuit Transient View for SCGs

The circuit transient view serves to improve the understandability of SCGs on hardware level. It models the SCG's program logic and can therefore be used for optimization issues. As the transformation takes place after sequentialized SCGs have been generated, all improvements, informations and optimizations of former transformation steps in the interactive compile chain are integrated into the circuit transformation step.

SSA SCG Transformation

After creating the meta-model, a transformation from sequentialized SCGs into circuits seems already possible by applying the mapping of Figure 4.5. But a closer look at the Sequentially Constructive Graphs (SCGs) to be transformed reveals data-flow problems. Those problems emerge if multiple assignments to one variable occur within one program. The main difficulty is to decide, which of the multiple assignments is the one to be used subsequently. As described by Johannsen [Joh13] SSAs are used to identify and resolve these data-flow problems.

Static Single Assignments As defined by Alpern, Cytron, Ferrante, Rosen, Wegman and Zadeck [AWZ88, CFR+91, RWZ88] a program is in SSA form, if each

```
x = 0;                          x_0 = 0;
y = 4;                          y_0 = 4;
x = y + 1;                      x_1 = y_0 + 1;
```

Listing 4.1. Transformation of assignments in simple program code into Static Single Assignments

```
y = 0;                          y_0 = 0
if(C) { y = 4; }                if(C) { y_1 = 4; }
    else { y = 9; }                 else { y_2 = 9; }
z = y;                          z_0 = phi(y_1, y_2);
```

Listing 4.2. A ϕ-function identifies the version of a variable to be assigned subsequently

variable is target of exactly one assignment in the program text. SSAs are used for program optimization such as redundancy analysis, detection of equality of variables or numeration of global variables.

Listing 4.1 illustrates the SSA form for a simple example in pseudo-code. Each variable `V` gets a new version number `V_i` for every new assignment targeting this variable, e. g., the variable is written to. Thus, every variable has a unique name and is only once targeted by an assignment. If a variable is used as part of an assignment, its latest version is used instead of the variable itself as shown in the third line of Listing 4.1.

In cases of multiple control-flow branches, such as invoked by `if-then-else` statements, a ϕ-function is used to determine which version of a variable should be assigned subsequently. As shown in Listing 4.2, the assignment to **`z_0`** depends on the result of the ϕ-function. If condition `C` it `true` **`y_1`** is assigned to **`z_0`**. Otherwise, **`y_2`** is assigned to **`z_0`**. This behavior emphasizes that the return value of the ϕ-function depends on the same condition as the decision which branch of an `if-then-else` statement is executed. Since the condition can either be evaluated to `true` or to `false` only one of the alternative execution branches is operated. Hence, an optimization for reducing the number of variables is possible. This optimization is illustrated in the Control Flow Graphs (CFGs) in Figure 4.10. Rather than using two different variables **`y_1`** and **`y_2`**, both execution branches assign to the same variable. That implies the assignment to **`z_0`** is now **`y_1`** instead of a ϕ-function as it was before optimization. Notice that now there are two assignments to **`y_1`** but only one of them will be executed before **`y_1`** is read.

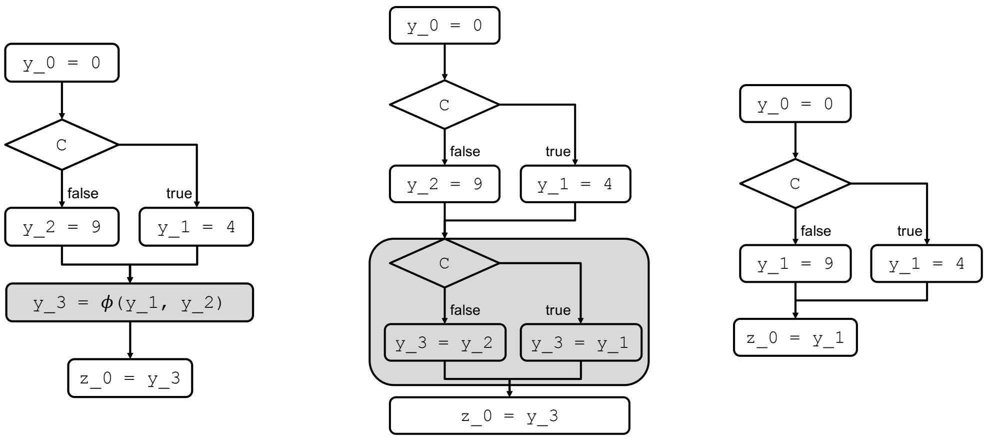


Figure 4.10. ϕ-function used for optimization in CFGs

Optimized SSA SCG

Preparative for the circuit transformation, one intermediate transformation of sequentialized Sequentially Constructive Graphs into SSA SCGs has been developed. In this step each relevant variable is replaced by its own SSAs and the ϕ-function is optimized as described above.

As mentioned in Section 4.1.4 the sequentialized SCG only consists of assignment and conditional nodes. More complex program constructs such as loops or goto statements are not taken into consideration for the transformation of sequentialized SCGs into SSA form.

Furthermore, not every variable occurring in the sequentialized SCG is relevant for the transformation. Merely, variables addressed by multiple assignments need to be transformed and are therefore relevant. In case of SCGs this means that output variables and input output variables are relevant. Since assignments to relevant variables in sequentialized SCGs always depend on a condition, the interesting parts for the transformation are the conditional nodes and their alternative branches. Conditional nodes in sequentialized SCGs have no assignment nodes on their `else-branches`. This means the SSA optimization step creates new assignment nodes which are positioned on the `else-branch` in the same sequential order as the associated assignments on the `then-branch`.

These new assignment nodes consist of an expression on the right side and a target of this expression on the left side. The target variable is named after the variable which is addressed by an assignment on the `then-branch`. Additionally, it gets a version number. This number is the highest version of the corresponding variable addressed by an assignment at this point of the execution. If the original variable on the `then-branch` was, e. g., named `V`, the variable on the `then-branch` might be `V_3` if this is the third time `V` is written to. This step highly simplifies the transformation into circuits. It is illustrated in Figure 4.12 for a simple program which emits a boolean output variable O when started. The original SCChart is depicted in Figure 4.11.

As there are two different kinds of relevant variables an important distinction needs to be made:

Input output variables are set at the beginning of each tick. They may be addressed by different assignments several times during a tick. The value of the last assignment within one tick will be emitted as output to the environment at the end of this tick. Hence, it is not necessary to store the output value of the variable as input for the next tick. In the optimization step of the SSA with the lowest version number the assignment node on the `else-branch` assigns the input output variable itself.

Output variables on the contrary get no input from the environment. Their output value for each tick has to be stored in a register as input for the next tick. This

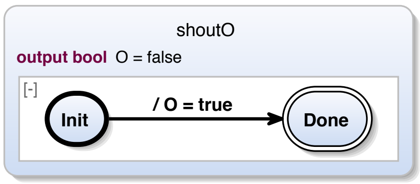


Figure 4.11. SCCharts model for a program which simply sets an output O to true in the first non–immediate tick.

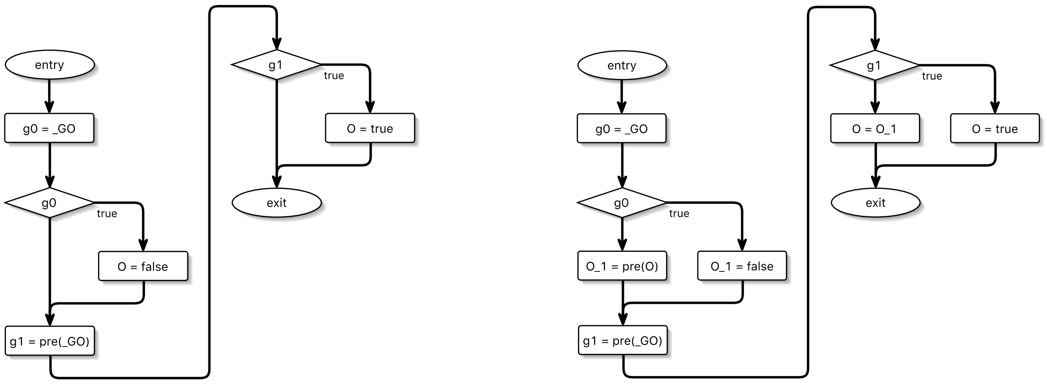


Figure 4.12. Transformation of sequentialized SCG into SSA SCG for an output variable O.

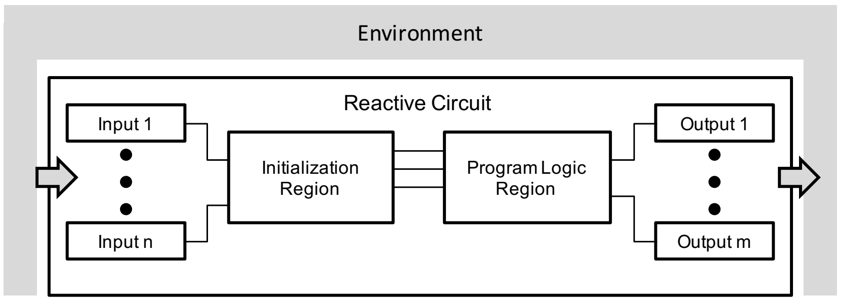


Figure 4.13. Organization of the circuit and its regions in the context of the controlled environment

is expressed by means of pre declarations. As shown in Figure 4.12 the lowest SSA version of output variable O is target of the assignment of `pre(O)`. This `pre(O)` value is the output value of O from the preceding tick. Moreover a `pre(V)` declaration for an output variable `V` is not transformed into SSAs since it is no new assignment to `V` but rather a storage container for the value of `V`.

Additionally, if an SSA variable is subsequently used on the right side of an assignment, this variable has to be replaced by the latest version of itself but no new version of this varibale is created. This assures that the right value is used.

Circuit Transformation

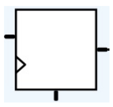
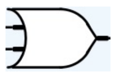
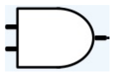
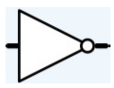
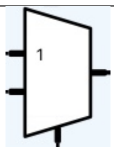
Step 5 in Figure 4.1 describes the circuit transformation. The circuit's structure is depicted in Figure 4.13. Transformed circuits are divided into two parts. The first part, the `Initialization Region`, provides the reset and tick logic and serves as initialization of the circuit. The second part, the `Program Logic Region`, is the translation of the information given by the SSA SCG and represents the program's logic. As the circuits are synthesized from SCCharts and therefore from a model of a reactive system, they are meant to be embedded in some kind of environment. This means, inputs from the environment are read and outputs for the environment are computed. Input variables and output variables as well as input output variables may appear in SCCharts. For each input variable and each input output variable one input port needs to be created. Likewise one output port is created for each output variable or input output variable. All local variables do not interact with the environment and therefore no ports are created for them.

Circuit Visualization

In order to visualize the circuits, for each kind of logic operation one logic gate is drawn as shown in Table 4.3. This small set of logic gates with additional wires is sufficient to create circuits for complex programs. The register's tick input gate is marked with a small triangle. Its reset input port is found at the bottom of the register. The number of input ports for OR and AND gates is not limited whereas NOT gates have only one input to be inverted. The implemented MUX have two input ports and one selection port at their bottom. The input port selected if the condition at the selection port evaluates to true is marked with a small 1. Links or wires are depicted as lines connected to ports.

To improve the readability of circuits, for each assignment node and its expression the circuit is divided into regions marked with red lines which contain the logic gates of the operation expressions.

Table 4.3. Visualization of logic gates

Register	OR-Gate	AND-Gate	NOT-Gate	MUX
				

Registers Registers are used to save values of variables within a circuit. This is necessary since the value of a simple wire may change within one tick whereas the output of a register is constant and may only change if a tick signal emerges. A register as depicted in Table 4.3 has three entry and one output port. The wire which holds the value to be saved in the register is applied to the first entry port. The tick signal is applied to the second port. Each time a rising edge from the tick signal emerges, the value applied to the first port is applied to the output port. The third entry port is the reset port. Each time the tick signal emerges and the reset signal is set, the register is reset. Registers marked with **FF** are sample registers. Those registers can not be reset.

In the course of the circuit transformation three different kinds of registers are created.

1. In the circuit initialization region, one register for each input variable and for each input output variable is created to ensure that input signals have a constant value throughout a tick.

2. For each pre declaration, a register in the program logic region is created. All Surface/Depth node constructs in SCGs result in such pre declarations. These declarations are used to symbolize that the stored system's state from the former tick is used.

3. The SSA SCG transformation needs a pre declaration for every output variable to save values for the next tick as pointed out in Section 4.2.3. Thus, a register in the program logic region is created for each of such pre declaration.

Circuit Initialization Region

The circuit initialization region enhances the circuit with those elements not necessarily needed to model the program's logic in hardware. It rather provides the circuit with means which guarantee the circuit to be in a stable state in each tick and particularly when reset. In words, even if the reset signal is present for more than one tick and one or more input signals alternate, the circuit's state and its output signals must not change.

In a simpler variant, as pointed out by Johannsen [Joh13], the computation of the circuit starts in the same tick as the tick the reset signal is present. That means the reset signal is equatable with the _GO signal. In this variant the input registers have no reset entry since their input signals need to be present in the reset tick. Thus, if the reset signal is present for more than one tick and an input signal alternates, this method incurs the disadvantage of changing output signals for each reset tick computed as the first tick.

To avoid this behavior the reset tick is separated from the start of computation. The _GO signal needs to be present one tick after the last tick the reset signal was present. Figure 4.14 depicts the reset logic in the initialization region of the AO program. The desired behavior is achieved by the following steps:

1. Add reset entries to the input registers. Since computation is meant to start one tick after the last tick the reset signal was present, alternating signals are not taken into account before start of computation.

2. Create a register to store the reset signal. This register has no reset entry for reset purpose but the reset signal as input to store. Each time the tick signal is present the output of this register tells if the reset signal is present.

3. Create a register to emit the _GO signal. This register will be placed behind the reset register. Thus, the output of the reset register is applied to this register as soon as no reset signal is present. The presence of the _GO signal is caused by the delay of the upstreamed reset register. If this is the first tick the reset signal is absent. At the beginning of this initial tick each register applies its input to

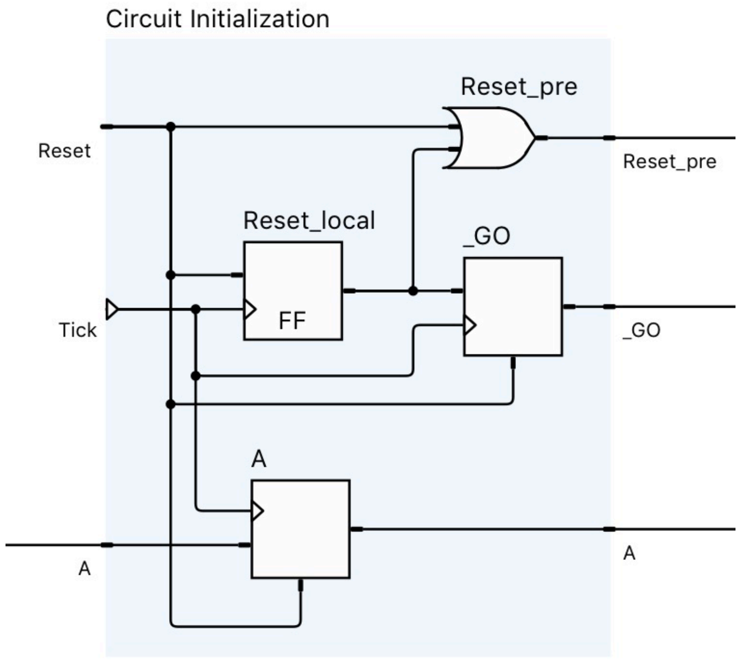


Figure 4.14. AO reset logic

its output. The `_GO` register and the reset register do that simultaneously. Since the reset signal was present in the last tick, the output of the reset register is still a present signal at the start of this tick. This present signal is the input of the `_GO` register and is applied to the `_GO` registers output. Thus, the `_GO` signal is emitted. The reaction computation starts.

4. To guarantee a stable state of the circuit, all `pre` registers need to be reset during the initial tick. Therefore the reset register's output and the reset signal are considered and put together in a logic `or` gate whose output is applied to the reset entries of all `pre` registers.

Figure 4.15 shows the signal curve for the described reset logic. The `pre` registers remain reset until they set their outputs in the second tick one tick after the `_GO` signal is set.

Program Logic Region

After the initialization region of the circuit has all necessary content the second region, the *Program Logic Region* as seen in Figure 4.13 needs to get its content. The content for this region is the complete circuit transformed from the beforehand

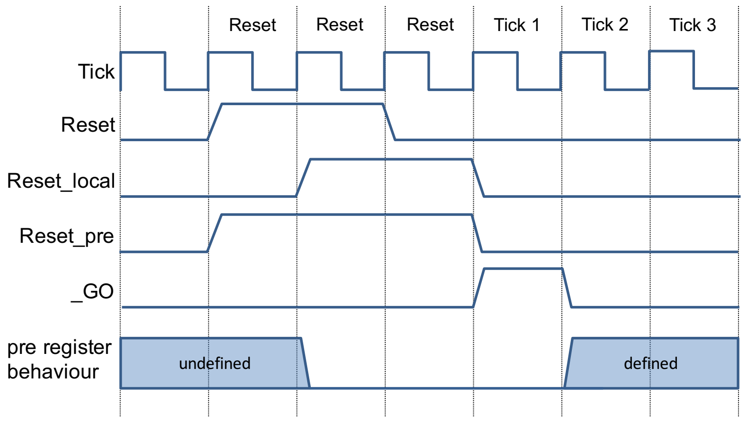
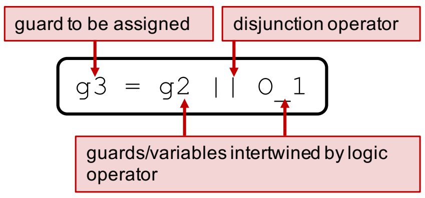
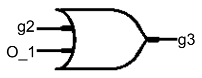


Figure 4.15. Signal curve of reset logic

created SSA SCGs. Since this SCG only consists of assignment nodes and conditional nodes a transformation of both components is exemplarily shown in the following.



(a) Assignment node

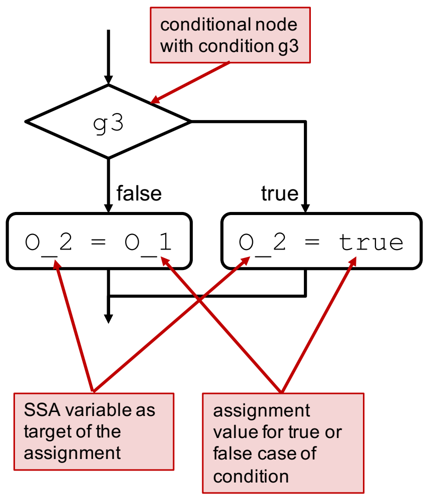


(b) Hardware translation for assignment node

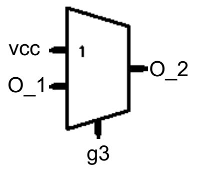
Figure 4.16. Hardware synthesis for assignment nodes

Assignment Nodes consist of a right and a left side. On the left side is the variable or guard addressed by the assignment. On the right side is the expression of the assignment. In Figure 4.16a the expression `g2 || O_1` is to be assigned to the guard `g3`. In this case the circuit transformation creates a new OR gate called `g3`. It has two inputs namely `g2` and `O_1`. Its output is `g3` as expressed the assignment node. In this case the variable `O_1` could be a SSA variable former known as `O` but changed to `O_1` in the SSA SCG transformation step.

In this example the operation is a disjunction. For other operations such as negation or conjunction, other associated gates are created. Figure 4.16b shows the corresponding hardware gate to the example assignment. If an expression on the right side of an assignment consists of more than one operation other gates need to be created and connected to each other according to those expressions. Particularly one input port for each for each subexpression is created and the subexpression itself invokes the creation of a new logic gate whose output is used as input for the gate created for the assignment.


conditional node with condition g3
g3
false
true
O_2 = O_1
O_2 = true
SSA variable as target of the assignment
assignment value for true or false case of condition

(a) Conditional node



(b) Hardware translation for conditional node

Figure 4.17. Hardware synthesis for conditional nodes

Conditional Nodes as shown in Figure 4.17a split the control-flow into two branches. The control-flow will follow only one of both branches which depends on the condition. If `g3` is present the condition evaluates to true and the `then-branch` is taken. Otherwise, if the condition evaluates to false the `else-branch` is taken. In the circuit transformation for each assignment node affected by a condition a Multiplexer (MUX) is created. It has two entries. One entry for the expression of the true case and one entry for the expression of the false case. Additionally, the MUX has one selection entry. This entry determines which of the incoming signals is applied to the MUX's output.

The MUX O_1 created according to the example applies `g3` to the selection entry and a constant `one` to the entry which is to be applied to the output if `g3` is present. The output of an other MUX namely O_1 is connected to the entry which is applied to the output if the condition evaluates to false. The hardware synthesis for this example is shown in Figure 4.17b.

4.2.4 Complete Transformation Example with AO

This section aims to illustrate the introduced transformations by means of an example. The AO program as depicted in Figure 4.18 has an input variable `A` and an output variable O. The program remains in the initial state `Init` until the input `A` is set to true. In this case the immediate transition to the final state `Done` is triggered and as a effect the output O is set to true. Since the transition is an immediate transition the program may terminate in the initial tick, if `A` is true.

Figure 4.19 shows the SCG and the SSA SCG for AO which has been introduced in Section 4.2.1. Finally Figure 4.20 shows the transformed circuit. In this view the tick wires are omitted for clarity. The initialization region on the left and the logic region on the right are expanded. The sequentialized SCG is not optimized regarding minimization of expressions. This is observable, e. g., at assignment node `g1 = g3 && A || _GO && A`. The sequentialized SCG does not use the minimal expression `(g3 || _GO) && A`. Since the circuits are transformed from sequentialized SCGs, they are as well not minimized.

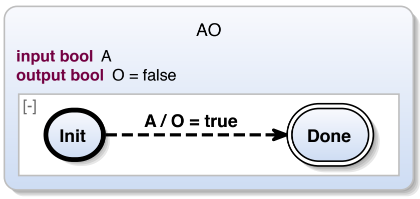


Figure 4.18. AO SCChart

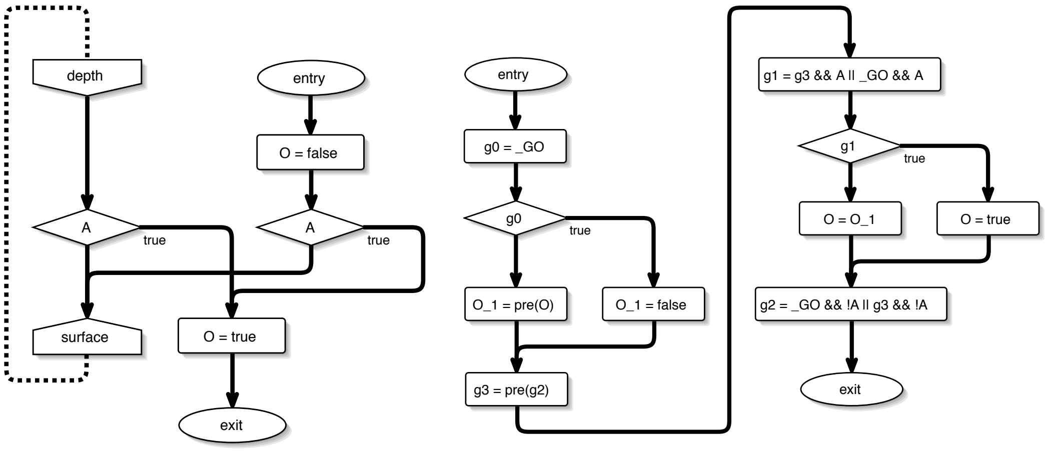


Figure 4.19. Left: AO SCG, Right: AO SSA SCG

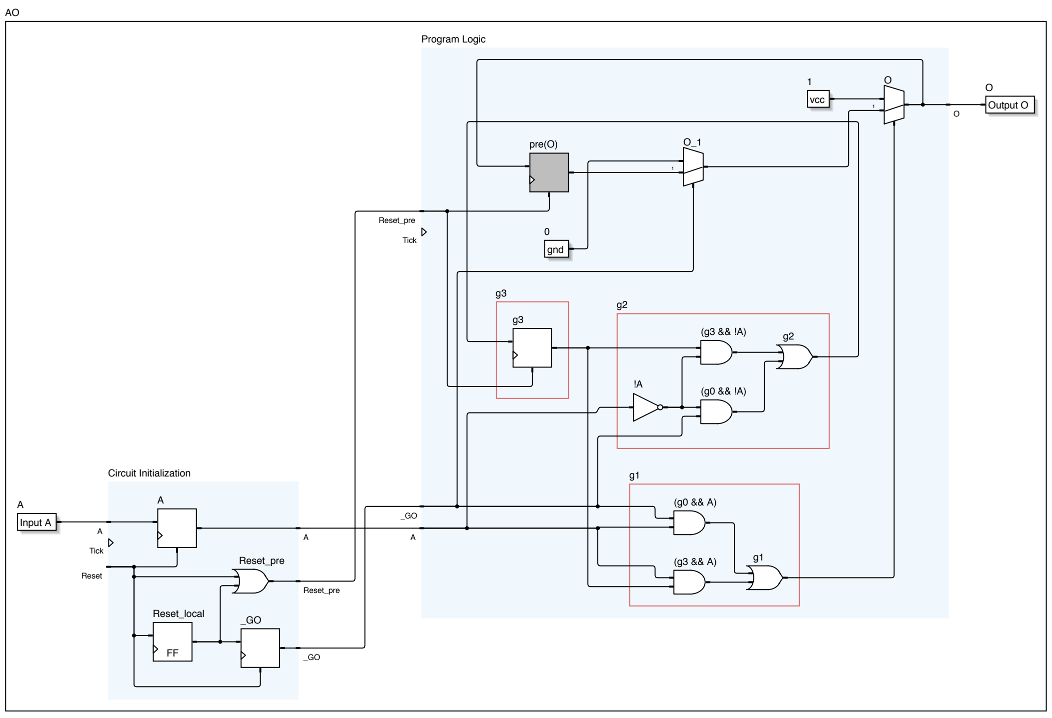


Figure 4.20. AO Circuit

4.2.5 Simulation

For reasons of comprehensibility and traceability, a simulation for the behavior of the transformed circuits was developed and implemented. This simulation corresponds with the simulation of the sequentialized SCG which already existed. In each tick the active guards are highlighted. Since assignments with guards and their expressions were translated into circuit components as described above, the corresponding logic gates can easily be highlighted in the circuit view. Thus, the understandability of how the program reacts to certain inputs is improved.

Figure 4.21 shows the highlighting of the simulation for the `AO` program. On the left hand side a sequentialized SCG is depicted. The program is not in its initial tick, otherwise the assignment node `g0 = _GO` would be highlighted. Instead, `g2` is highlighted since the input variable `A` is not true. `g3` saves the true value of `g2` for the next tick and is therefore also highlighted. As long as `A` remains false this program state will not change. In the sequentialized SCG on the right hand side `A` has finally been set to true. Since `g3` saves the true value of `g2` from the last tick, `g1` is now highlighted and set to true. Hence, the conditional node `g1` evaluates to true and the output `O` is set to true.

Figure 4.22 shows the same tick as the right SSA SCG in Figure 4.21. The

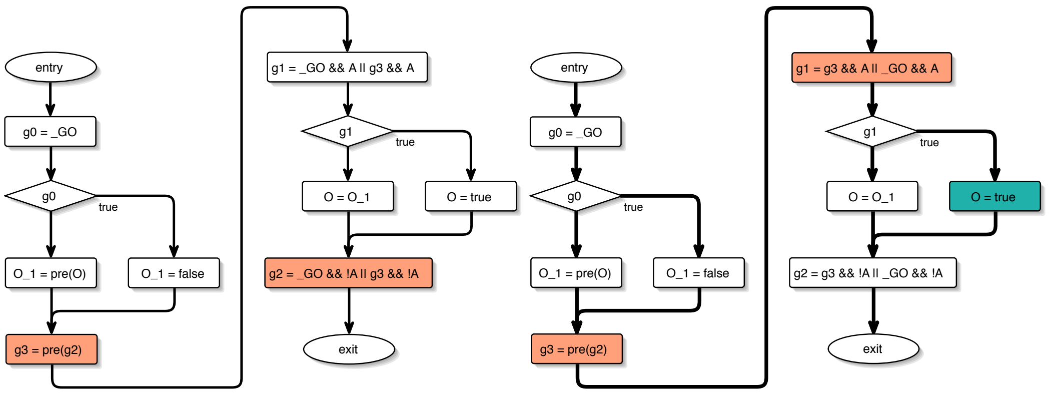


Figure 4.21. Simulation visualization of AO SSA SCG. Active guards and their assignment nodes are highlighted. Left: Non initial tick and the boolean input A is not set to true. Right: One tick later A is set to true.

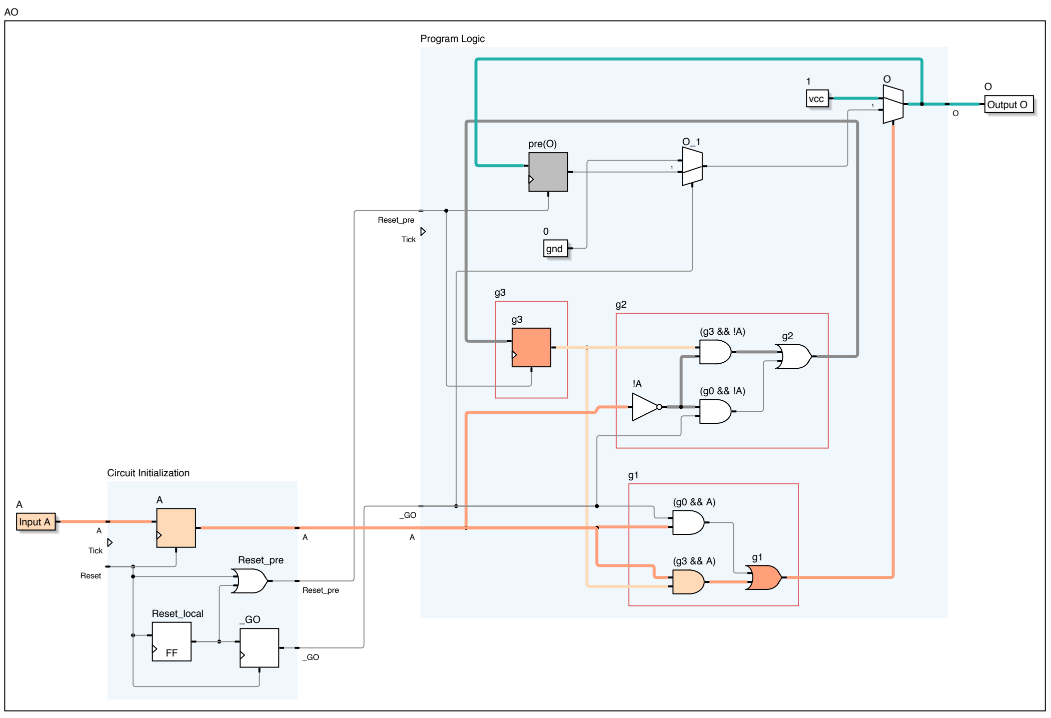


Figure 4.22. Simulation visualization of AO circuit. The to the SSA SCG corresponding logic gates are highlighted. In this tick A has been set to true.

orange color is reserved for signals which change from 0 to 1. As shown, the circuit behaves as desired and the input A set to true implies the output O to be true. Green highlighted wires mark that a MUX applies the value from its input port 1 to its output. That means, the selection port input was true. Hence, the corresponding conditional node in the sequentialized SCG was evaluated to true. Notice that a green highlighted wire does not necessarily imply a true signal or voltage on the wire.

4.2.6 Complete ABRO Example

To exemplify the circuit transformation and its simulation working for more complex programs than AO, the ABRO example is fully transformed into a circuit and simulated. Figure 4.23 shows ABRO, the "Hello World!" of SyncCharts, as SCCharts on the left side and its corresponding SSA SCG on the right side. Figure 4.24 shows the program logic of the corresponding circuit. The initialization region as well as all tick wires are omitted due to space limitations.

ABRO's Behavior: The ABRO SCChart has three boolean inputs A, B and R and one boolean output O as shown in the declaration interface. The initial state ABthenO and the state WaitAandB as well as the initial states wA and wB are entered when the program starts. The regions HandleA and HandleB run concurrently in WaitAandB. Initially when state ABthenO is entered, the output O is set to false. This happens each time this state is reentered. In region HandleA the program remains in state wA until the input A is set to true. In that case it transitions to the final state dA. Region HandleB provides the same behavior for input B. If both final states are reached, the termination transition to the final state done is immediately taken and O is set to true. However, if the reset signal R is present at a time during execution the state ABthenO is preempted and the self strong preemptive transition (red dot) is taken. ABthenO is reentered. This transition is taken even if in the same tick the immediate transition to done would have been taken. O is reset to false and the program reenters the initial states wA and wB.

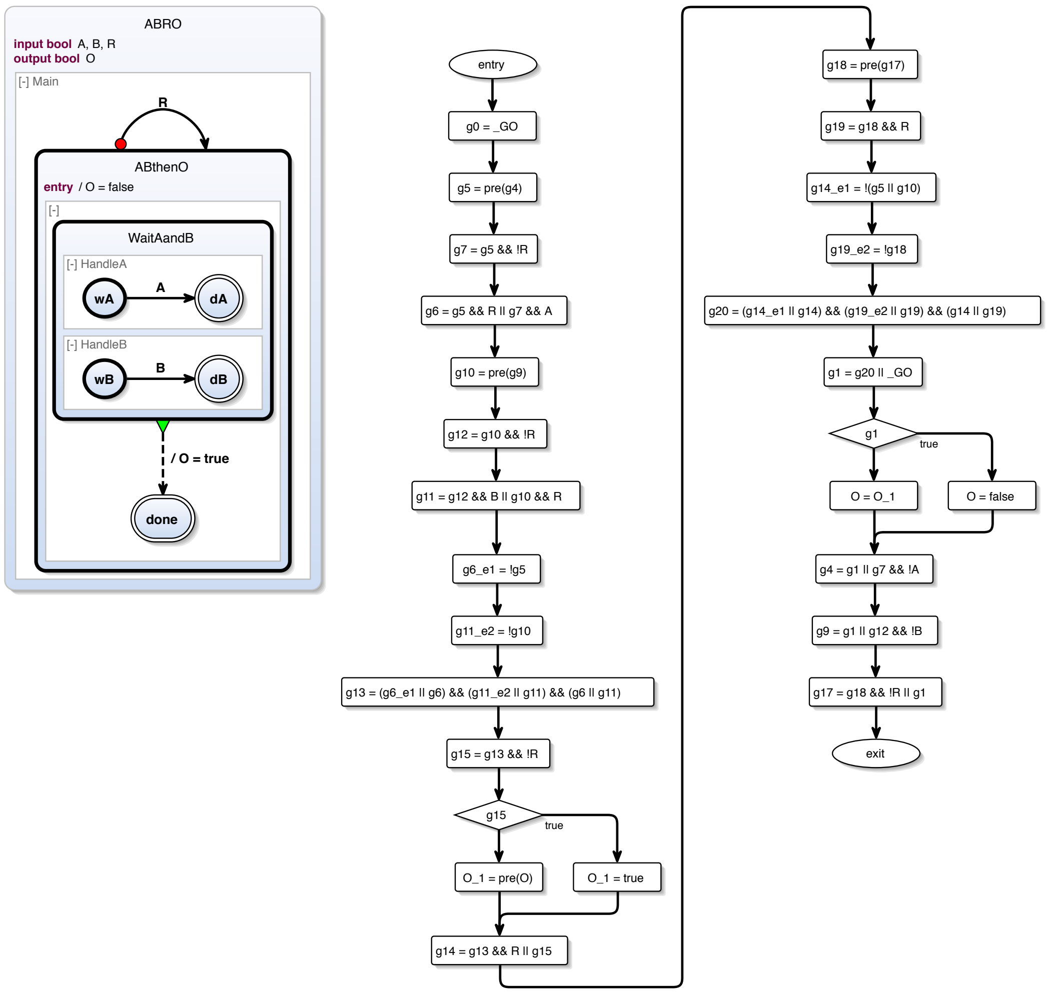


Figure 4.23. ABRO – the "Hello World!" of synchronous programing. Left: ABRO SCChart. Right: ABRO SSA SCG

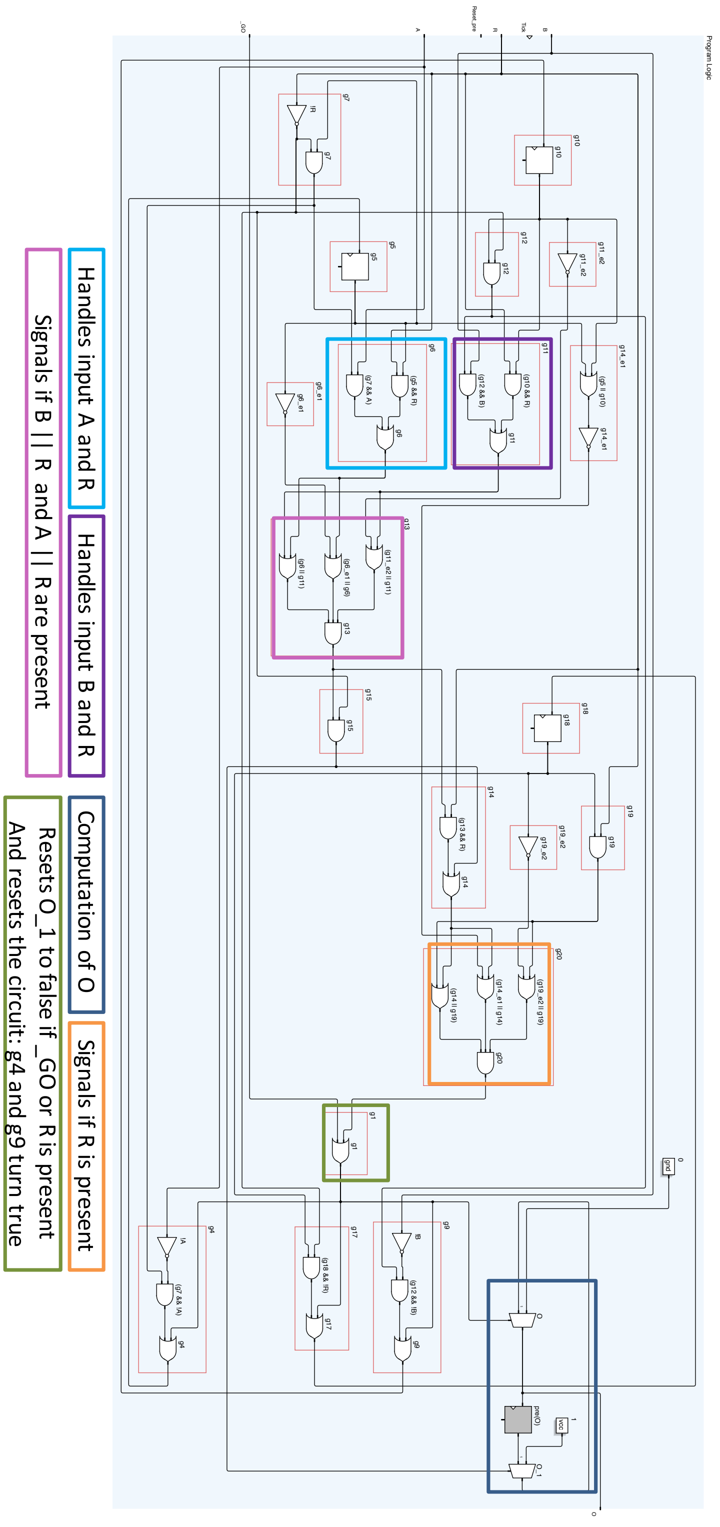

Program Logic
Handles input A and R
Signals if B || R and A || R are present
Handles input B and R
Computation of O
Signals if R is present
Resets O_1 to false if _GO or R is present
And resets the circuit: g4 and g9 turn true

Figure 4.24. Annotated ABRO circuit. Only program logic region is shown.

Chapter 5

Implementation

To validate the proposed interactive incremental hardware synthesis approach, I have implemented the corresponding SLIC transformations and the circuit diagram synthesis in KIELER based on a set of Eclipse plug-ins. Figure 4.1 describes the components and how they integrate into the KIELER project. The circuit meta-model provides the basis for the circuit transformation and the visualization in KLighD. A detailed description of the meta-model and its components can be found in Section 4.2.2 and is not further discussed in this chapter.

All transformations are implemented using Xtend and are model-to-model transformations. This means the transformed models have a source and a target meta-model. If source and target meta-model are the same, the transformation is called *inplace* [Mot09] transformation. The first introduced transformation from sequentialized SCG into SSA SCG is such an inplace transformation.

5.1 Plug-in Overview

Table 5.1 gives an overview of the implemented classes and the containing plug-ins. The circuit meta-model and all generated code for the classes and interfaces contained in the meta-model are located in the **`de.cau.kieler.circuit`** plug-in. Since the plug-ins **`de.cau.kieler.circuit.edit`**, **`de.cau.kieler.circuit.editor`** and as well the plug-in **`de.cau.kieler.circuit.test`** have been automatically generated, they are not part of this chapter.

The implemented transformations from sequentialized SCG into SSA SCG and the circuit transformation are located in the **`de.cau.kieler.scg.circuit`** plug-in. Both of which are part of the *Circuit Feature Group* in KiCo and are discussed in Section 5.2. The transformation into SSA SCG is an *inplace* model transformation since the SCG meta-model is target and source model of the transformation. The circuit transformation whereas is a transformation from SCG meta-model in circuit meta-model.

The **`de.cau.cs.kieler.circuit.klighd`** contains the visualization component for circuits. In this plug-in the class *CircuitDiagramSynthesis* initiates the transformation resulting in a KGraph model which is visualized. Section 5.3 covers

Table 5.1. Overview of implemented plug-ins and functionalities

Plug-in	**Content Description**
de.cau.kieler.circuit	The Ecore file **circuit.ecore** contains the circuit meta-model.
	All generated Java classes of the model's object are found in the **de.cau.cs.kieler.circuit** package.
de.cau.cs.kieler.scg.circuit	The class **SeqSCG2SSA_SCGTransformation** transforms the sequentailized SCGs into SSA form. (1)
	The class **SSA_SCG2CircuitTransformation** transforms the SSA SCGs into circuits. (2)
	The class **CircuitInitialization** adds input and output ports to the circuit and creates reset logic and registers for the *Initialization Region*.
	The class **LinkCreator** adds all links to the circuit's regions.
de.cau.cs.kieler.circuit.klighd (3)	The class **CircuitDiagramSynthesis** starts the transformation of circuit meta-model elements into KGraph elements.
	The class **ActorSynthesis** calls a synthesis for a specified gate type depending on the actor type which is to be transformed.
	The interface **IDrawableActor** ensures all by the ActroSynthesis called gate synthesis have a method to actually draw the specified gate.
de.cau.cs.kieler.circuit.kivi (4)	The class **CircuitVisualizationDataComponent** is responsible for the highlighting of circuit elements when the simulation is started.

this synthesis. The simulation is located in the **de.cau.cs.kieler.circuit.kivi** plug-in. It is discussed in Section 5.4.

5.2 Circuit Feature Group Transformations

The SCG to SSA SCG transformation (1) as well as the SSA SCG to circuit transformation (2) are located in the **`de.cau.cs.kieler.scg.circuit`** plug-in. Both of them are implemented as *one-pass-transformations* meaning only one iteration through the source model is necessary for the complete transformation into their target models.

5.2.1 SCG to SSA SCG Transformation

This intermediate transformation step (1) has been introduced in Section 4.2.3. It is a model-to-model transformation from the SCG meta-model into the same. As conceptually described for each new write to an output variable, a new version of this variable is created and replaces the former target variable. Furthermore, all subsequent references to this variable are replaced by its latest version. Figure 5.2.1 shows the assignment node and conditional node example shown in Figures 4.16a and 4.17a as seen from the perspective of the implementation. This transformation searches for the `entry` node and uses it as starting point. Since the input of the transformation is a sequentialized SCG the `entry` node is unique. Beginning with this `entry` node, the transformation follows the control-flow and calls either a method to transform assignment nodes or a method to transform conditional nodes and those nodes influenced by the conditional node. A sequentialized SCG has only assignment and conditional nodes and one entry and one exit node. Listing 5.2.1 shows the iteration through the SCG. Transformation methods are called depending on the node type.

Before the transformation takes place in case of conditional nodes and the subsequent forking control-flow, no assignment nodes on the `else-branch` exist. Hence, for each assignment node on the `then-branch` a new assignment node on the `else-branch` is created. Further on a new *valued object* is created. This valued object has the name of the target variable of the assignment plus a version number. The left side of Figure 5.2.1 depicts the described case. Before transformation the `else-branch` pointed directly to the end of the fork and the assignment node of the `then-branch` contained `O = true`. As `O` happens to have already been written before the new version number is 2. If the condition is evaluated to false it is desired to ensure that the older version, in this case `O_1`, is assigned to `O_2`. Therefore the *valued object reference* to `O_1` is assigned to `O_2` as an expression. If the `then-branch` itself contains a conditional node, the same procedure is called recursively.

The right side of Figure 5.2.1 depicts the situation in which the left side of the assignment node is not an output variable which has to be replaced.

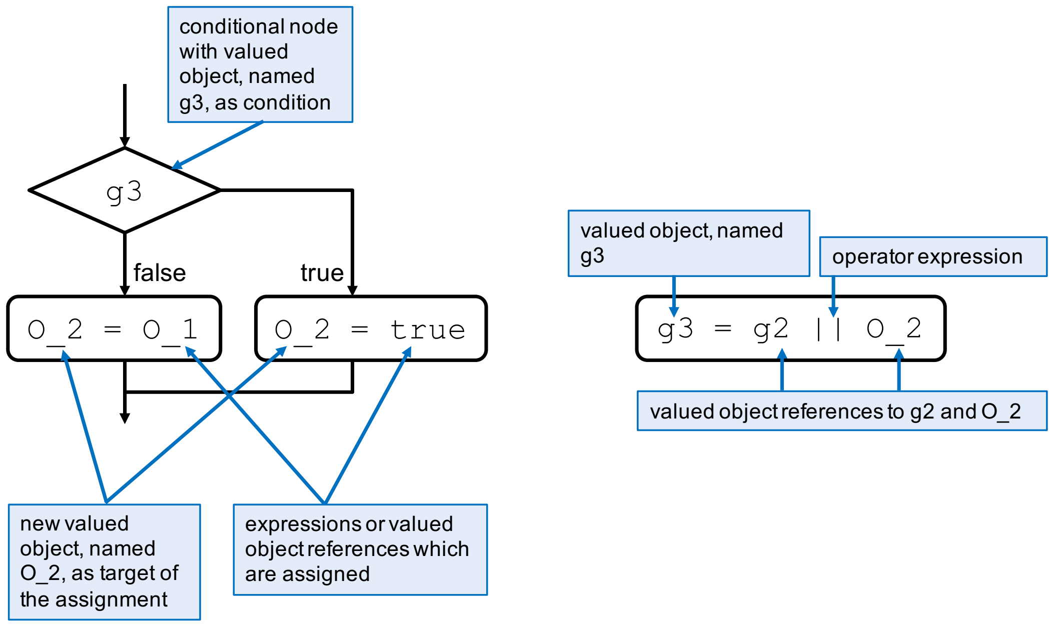

conditional node with valued object, named g3, as condition
g3
false
true
O_2 = O_1
O_2 = true
new valued object, named O_2, as target of the assignment
expressions or valued object references which are assigned
valued object, named g3
operator expression
g3 = g2 || O_2
valued object references to g2 and O_2

Figure 5.1. Conditional node (left) and assignment node (right) after SSA SCG transformation.

Comparatively, the expression on the right side of the assignment contains such a variable. Before transformation, this assignment node contained `g3 = g2 || O`. Since the input of this transformation is a sequentialized SCG, it simply follows the control flow. Thus, the highest version of every SSA variable can be stored and the former created valued object for this version is referenced if an output variable emerges in an expression. Each time a new valued object is crated for the same variable, it replaces the stored valued object associated with this variable.

5.2.2 SSA SCG to Circuit Transformation

In Section 4.2.3 the circuit transformation is introduced conceptually. This transformation is a model-to-model transformation from the SCG meta-model into the meta-model for circuits. Before the transformation of the SSA SCG takes place, three actors are created:

1. One root actor as the complete circuit region containing all nodes for inputs and outputs, 2. the circuit initialization region and 3. the program's logic region. The initialization region is filled with gates as described in Section 4.2.3. For the transformation of the program's logic, again, the `entry` node is searched and used as a starting point. Similar to the SSA SCG transformation, this transformation follows the control flow beginning with the identified `entry` node. Since the SSA

```
def void createSSAs(Node n, SCGraph scg) {
    if (!(n instanceof Exit)) {
        if (n instanceof Assignment) {
            transformAssignmentNodes(n, scg)
            createSSAs(n.next.target, scg)
        } else if (n instanceof Conditional) {
            val target = n.^else.target
            transformConditionalNodes(n, n, n, n.^else.target, scg)
            createSSAs(target, scg)
        }
    }
}
```

Listing 5.1. Extract from SeqSCG2SSA_SCGTransformation. Iteration through sequentialized SCG following the control-flow and calling transformation methods for assignment and conditional nodes.

```
def void transformNodesToActors(Node n, Actor logic) {
    if (!(n instanceof Exit)) {
        if (n instanceof Assignment) {
            transformAssignment(n, logic)
            transformNodesToActors(n.next.target, logic)
        } else if (n instanceof Conditional) {
            transformNodesToActors(transformConditionalNodes(n,
                                    n.then.target, n.^else.target, logic), logic)
        }
    }
}
```

Listing 5.2. Extract from SSASCG2CircuitTransformation. Iteration through SSA SCG following the control-flow and calling transformation methods for assignment and conditional nodes.

SCG contains the same node types as the sequentialized SCG, only assignment and conditional nodes need to be distinguished. Listing 5.2.2 shows the iteration through the SSA SCG. Depending on the node type, transformation methods are called. After creating all gates and their ports they need to be connected via links. For each region of the circuit the *LinkCreator* class is executed. This class creates links in each region by connecting ports with equal names with each other. Ports can have different types: `"In"`, `"Out"`, `"InConnector" + nameOfRegion`, `"OutConnector" + nameOfRegion`. In each region different types of ports may be connected as shown in Table 5.2. For all links, the connected ports must not be contained by the same gate.

5.3 KLighD Circuit Diagram Synthesis

For actually visualizing circuits yet another model-to-model transformation (3) from the circuit meta-model into the KGraph meta-model is necessary. This transformation extends the *AbstractDiagramSynthesis* provided by KLighD. With this, a *transform* method needs to be overwritten. This method returns a `KNode` as root object of the subsequently transformed KGraph model. Furthermore, it presumes a root object of the source model which is to be synthesized. In case of the circuit visualization this root object is an `Actor` object. Iterating through the circuit, each `Actor` object is transformed into a `KNode`, each `Link` object is transformed into a `KEdge` and each `Port` is transformed into a `KPort`. All transformed objects are associated with their source circuit objects which is an important step for further usage, such as in the simulation. Listing 5.3 shows the synthesis for a NOT gate.

Each actor of the circuit has a `type` parameter. According to this parameter the visualization transformation links all information of how the visual representation shall look like to the `KNode` object which is associated to the actor. This is done by classes for each circuit element type which hold the drawing informations. For example one of this classes is *AndActorSynthesis*. This class holds the information how a logic and gate is to be drawn in the circuit visualization. The method *draw* is inherited from the interface *IDrawableActor* which has to be implemented by each class used to depict a circuit element.

Table 5.2. Valid links between port types for each region.

Initialization Region	`"Out" – "In"`
	`"InConnectorInit" – "In"`
	`"Out" – "OutConnectorInit"`
Program Logic Region	`"Out" – "In"`
	`"InConnectorLogic" – "In"`
	`"Out" – "OutConnectorLogic"`
Reactive Circuit Region	`"Out" – "InConnectorInit"`
	`"OutConnectorInit" – "InConnectorLogic"`
	`"OutConnectorLogic" – "In"`

5.4 Simulation Visualization

The simulation (4) is implemented as *DataComponent* and integrated into KIEM. As a DataComponent the circuit visualization class needs to supply at least the following five methods:

1. `initialize()`: This method is called before the simulation begins. In the circuit simulation, this phase is used to fill up several maps with data. Mainly, the drawing information of actors are linked to the associated actor's names. These informations are later used to highlight the circuit components according to the names of active guards. Those names are the same as the names of the active gates or registers in the circuit.

   Additionally, in this phase all wires of the circuit are grayed out since no wire has any signal on it.

2. `wrapup()`: If the simulation is stopped by the user, this method is called. It

```
override draw(Actor actor) {
    val KNode node = actor.node
    node.setNodeSize(30, 30);
    node.addRectangle => [
        it.invisible = true
        it.addPolygon => [
            it.id = "highlightable"
            it.lineWidth = 1
            it.lineCap = LineCap.CAP_ROUND;
            it.lineJoin = LineJoin.JOIN_ROUND;
            it.background = "white".color;
            it.selectionBackground = "gray".color;
            it.addKPosition(LEFT, 0, 0, TOP, 1, 0)
            it.addKPosition(RIGHT, 2, 0, TOP, 0, 0.5f)
            it.addKPosition(LEFT, 0, 0, BOTTOM, 1, 0)
        ];
        it.addEllipse => [
            it.id = "highlightable"
            it.setBackground("white".color).lineWidth = 1;
            it.setAreaPlacementData.from(LEFT, 24, 0, TOP, 12, 0)
                                    .to(RIGHT, 0, 0, BOTTOM, 12, 0);
        ]
    ];
    return node;
}
```

Listing 5.3. Extract from NotActorSynthesis.xtend. *draw* method for NOT gate.

provides clean-up code which removes all kinds of highlighting from the circuit visualization.

3. `step()`: After the simulation has started and the *initialize* method has finished, the user may prompt as many steps as wanted. Each time a step is prompted, this method is called. This method gets a JSONObject which holds the data of which guards are to be highlighted in this step. Since the circuits are transformed from SCCharts, each step can be assumed as a *tick*. Hence, the simulation highlights the active gates according to the active guards in each tick. Furthermore, all outgoing wires of active gates are highlighted and thereby subsequently activated gates and wires are highlighted as well.

4. `isProducer()`: Since the circuit visualization produces no data for other DataComponents, this predicate returns `false`.

5. `isObserver()`: This predicate returns `true` since the circuit DataComponent observes data from the JSONObjectDataComponent.

Chapter 6

Evaluation

In this chapter the transformed circuits are analyzed for correct behavior. The synthesized circuits are compared to circuits generated in former works. The behavior is validated with means of the simulation. Furthermore, the scaling of circuits depending on the number of nodes in sequentialized SCGs is evaluated.

6.1 Comparing ABO

As first validation method the ABO circuit extracted from Johannsen's thesis [Joh13] was compared to the newly generated ABO circuit of this thesis. Figures 6.2 and Figure 6.1 depict both circuits. In both figures, the inputs are on the left side and the outputs are on the right side. Both circuits have been generated from SCCharts. Whereby Johannsen worked with a former implementation of SCCharts. Furthermore, Johannsen used the Sequentially Constructive Language (SCL) which is the textual representation of SCGs as basis for his transformation. The incremental interactive approach introduced in this thesis uses SCGs as basis. The frames colored similarly represent the equivalent translated components of the circuits. As shown, both circuits use several MUXs for computation of the output values. The green marked areas depict the parallel running regions of ABO. One of which depicts the thread waiting for A. The other one depicts the thread waining for B. Their `join` is expressed as `and-gate`. The gray registers in Figure 6.2 do not occur in Figure 6.1 since they have been implemented in the *CircuitInitialization* region which is not depicted since it does not express any program logic. The red marked register holds the _GO signal in both circuits. Since Johannsen's circuits are transformed from a former version of SCCharts a few gates differ comparing the circuits. For example while Johannsen's circuit has only one MUX for `B` the newly generated circuit has two. That means one MUX for each new assignment to `B` which can be compared to the SSA SCG. Additionally, the new circuit has a logic which is activated in the case `A` is set to true, namely the area g2. The area g3 describes the logic for the case `A` is not set to true. This strict distinction and aggregation of gates in areas is an improvement compared with Johannsen's circuit.

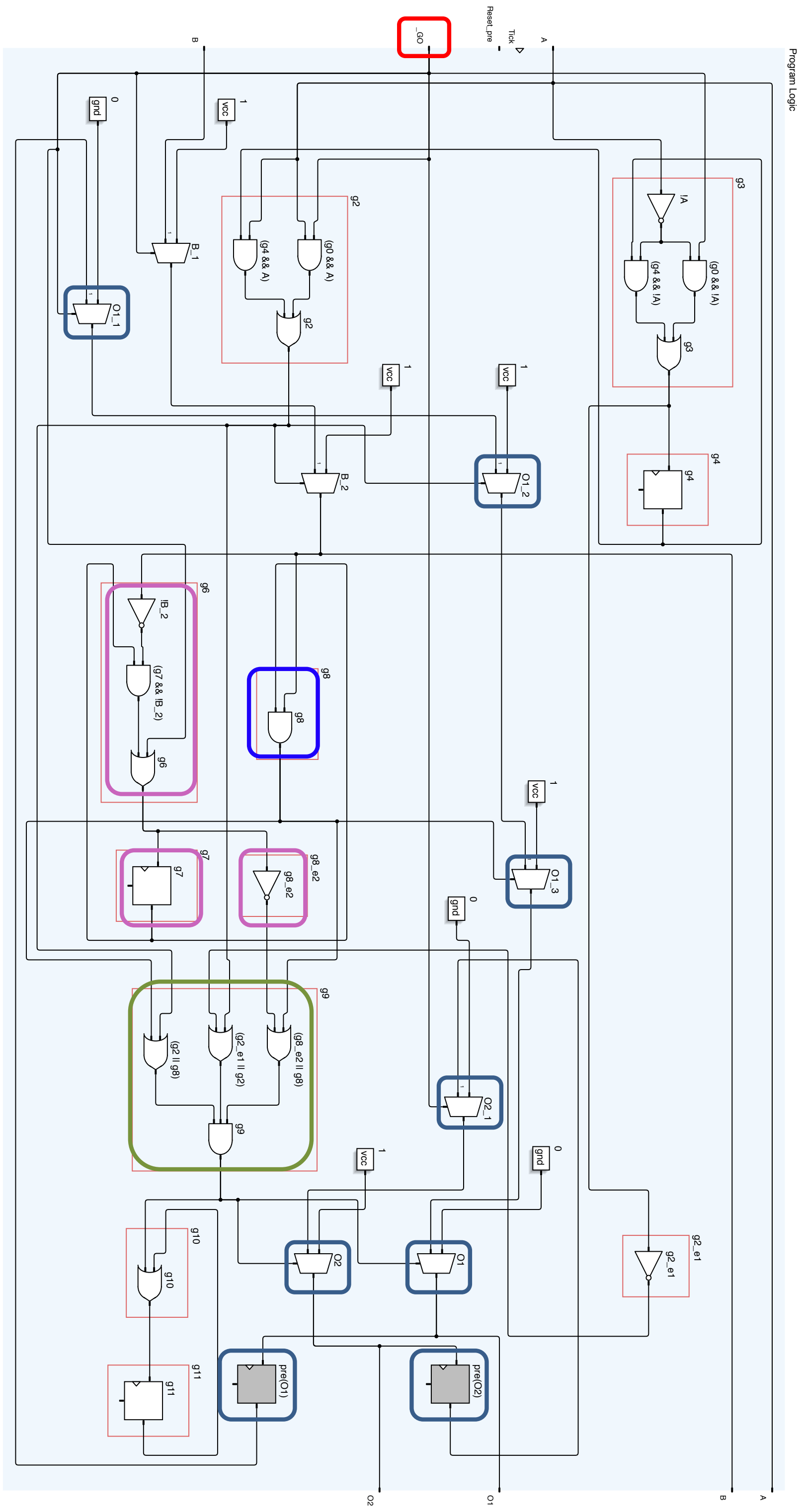


Figure 6.1. The ABO example transformed with the introduced transformation

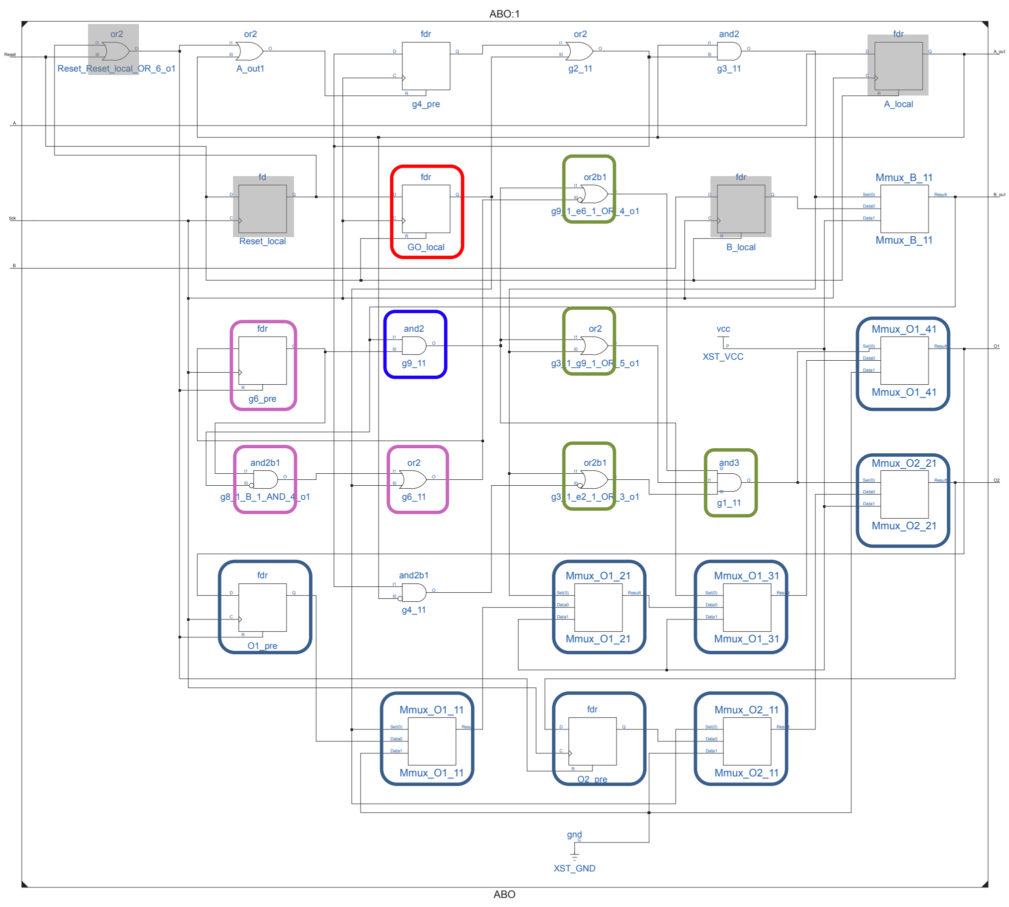


Figure 6.2. The ABO example transformed from SCCharts [Joh13]

6.2 Simulation of AO

To validate the correct behavior of the generated circuits AO has been simulated and tested with two execution traces. In the following the different highlighting of the SSA SCG and the circuit are depicted for each tick in the two traces. The first of which is simply a test for the immediate transition in AO. That means in the very first tick `A` is present and O should be set to true as can be seen in Figure 6.3. In the second example `A` is not present in the first three ticks and is set to true in the fourth tick. Figure 6.4 shows the first four ticks of this simulation.

In this thesis the simulation of AO is exemplarily depicted for validation purposes since AO is not as complex as ABRO or ABO. However, the latter have been tested with various execution traces and both circuits delivered the expected outputs for defined inputs and ticks.

In the first example the simulation shows that `A` set to true in the first tick triggers the output O to be true at the end of the tick. The *or-gate* g1 selects the green marked input of the MUX O to be applied to its output.

In the second example in the first tick the output O is set to false caused by the _GO impulse g0 which applies the green marked value – in this case 0 – to the output of MUX O_1. The gates in region g1 are not activated until an input `A` is set to true. Up to this point of the execution solely the gates of region g2 are activated in each tick waiting for an input `A` and remembering to restart the request for an input `A` to be true with help of the register g3. Eventually, in the fourth tick `A` is set to true and hence the output O is set to true.

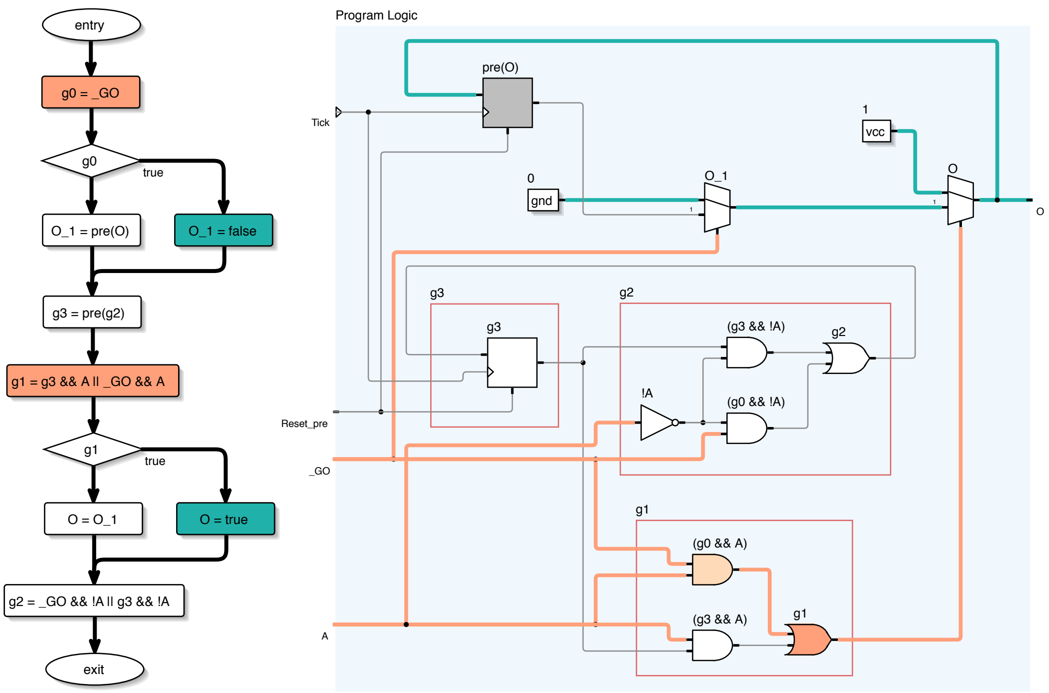


Figure 6.3. Examplary simulation of AO. A is set to true in the first tick.

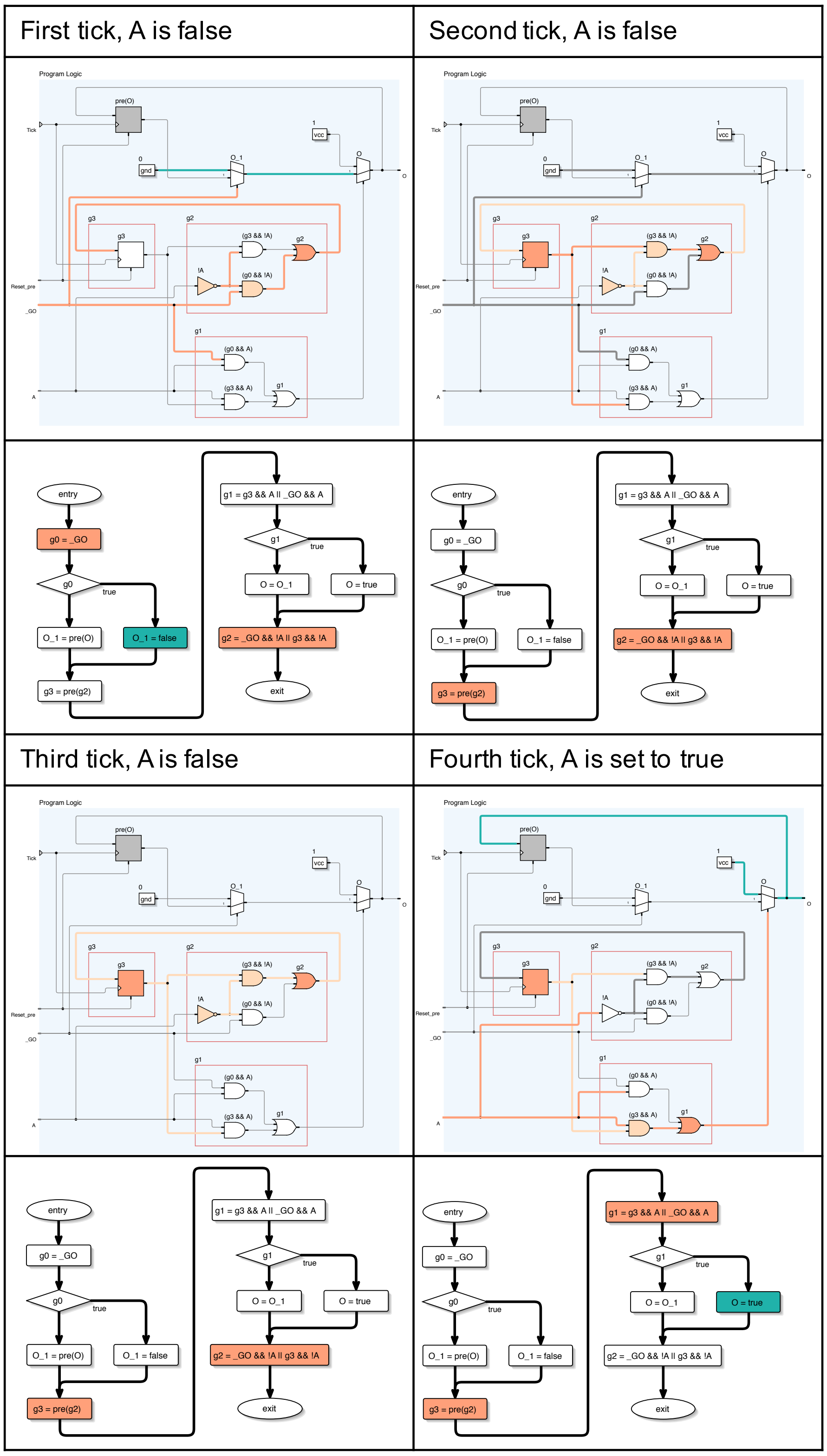


Figure 6.4. Examplary simulation of AO. A is set to true in the fourth tick.

6.3 Validation Improvements

The simulation introduced in this paper uses the C code generated from the sequentialized SCG to highlight gates and wires. This approach assumes the user to validate whether the circuit reacts as expected. More precisely, the user needs to compare inputs and outputs of the circuit to the inputs and outputs of the corresponding SCG or SCChart.

An enhanced solution is to translate the circuits back into an SCGs and compare the ESO files as described in the following and in Figure 6.5. First of all, an ESO file is generated by creating outputs for certain inputs for SCCharts models. Those outputs are gained by the transformation into SCGs and C code which is executed in KIEM. In case a transformation from circuits back to SCGs exists, C code may be generated out of these SCGs and hence outputs and ESO files can be compared.

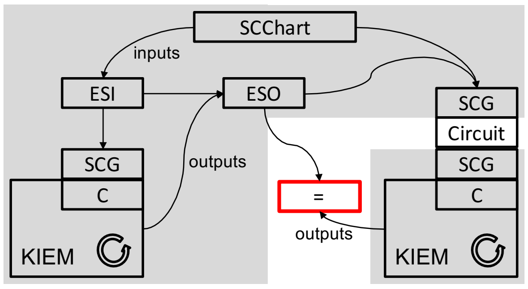


Figure 6.5. Validation of the circuit transformation with ESO files

This validation method is extendable if the ESO file is not generated from SCCharts but from Esterel. Again the Esterel–based ESO file and an ESO file generated from SCGs which are re-generated based on circuits may be compared.

A third approach would be to translate the circuits in a Ptolemy model and to execute a simulation in this tool.

6.4 Scaling

To validate the circuit transformation scales well, different models have been tested. For this purpose five different programs have been modeled. The programs sorted by number of nodes in ascending order are AO, ABO, ALDO, ABRO, Elevator and DVDPlayer. The nodes are counted for normalized SCCharts, SCG, sequentialized SCG, SSA SCG and Circuits. In the following, the number of nodes in circuits is measured in the program logic region omitting multiple occurrences of `gnd` and `vcc` gates.

Figure 6.6 shows the size of circuits depending on the corresponding SSA SCG's node size. In normalized SCCharts the number of nodes is between 5 for the AO model and 86 for the DVDPlayer model. The number of nodes in the sequentialized SCG and the SSA SCG is in all models almost the same. This is not surprising, since merely a few assignment nodes on conditional `else-branches` are added for the case the condition evaluates to false. The number of nodes in the SSA SCG reaches from 12 in AO to 155 in the DVDPlayer model. Focusing on the circuits, it is observable that in no case the number of nodes exceed twice the number of nodes in the SSA SCGs. The AO circuits has 11 nodes and the DVDPlayer has 200 nodes.

There are three different aspects which influence the scaling of circuits depending on the nodes in sequentialized SCGs:

1. Expressions like `gX = gY` are simply translated as one wire with two different names and therefore do not increment the number of nodes in circuits.

2. Guard expressions like `gX = gY || gZ` produce as many logic gates as nested operator expressions exist. `gX = gY || gZ` or `gX = pre(gZ)`, e. g., produce only one gate, `gX = (gY || gZ ) && A`, e. g., produces two gates. Notice that the `pre` operator results in the creation of a register which stores. Guard expressions can grow very large depending on the number of concurrent running regions in SCCharts. The joining guards tend to have expressions with multiple operation expressions. While those joining guards result in only one assignment node in the sequentialized SCG. However, large expressions in the SCG result in a high number of logic gates in the circuit.

3. The new assignment nodes on `else-branches` in SSA SCGs do not double the number of created logic gates in circuits, since only one MUX is created. Each MUX summarizes two assignment nodes: One from the `then-branch` and one from the `else-branch`. This is the reason why, e. g., the ALDO circuit has less nodes than the sequentialized SCG.

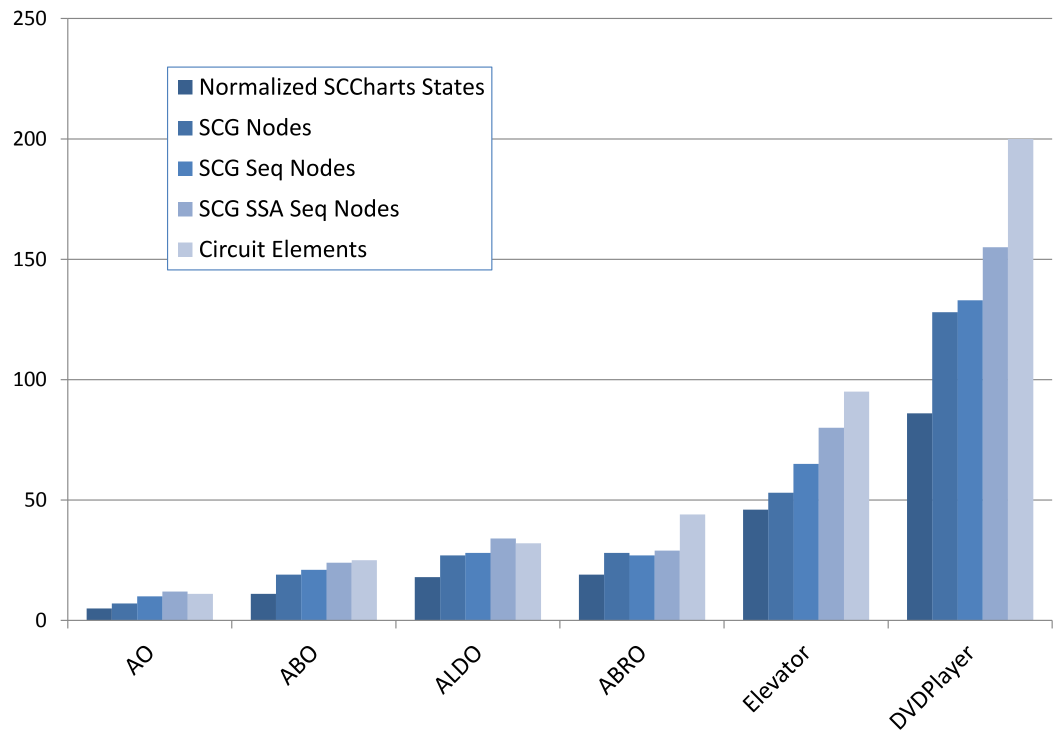


Figure 6.6. Scaling of synthesized circuits compared to the corresponding SSA SCG and SCCharts depending on the number of nodes.

Chapter 7

Conclusion

This chapter summarizes the proposed interactive incremental hardware synthesis, the developed implementation and evaluation results. It further checks again the requirements in the initial problem statement. Finally, possible directions for future work are suggested.

7.1 Summary

This thesis introduces a hardware synthesis from SCCharts adopting the interactive incremental approach as implemented in the KIELER SCCharts tools [MSvH14]. All implemented transformations have been integrated into KIELER and are available as feature transformation steps in the compilation chain.

The sequentialized SCGs introduced by Smyth [Smy13] has been the starting point for the introduced approach. The sequentialized SCG has been chosen due to several reasons.

(a) The parallel execution of the transformed circuits needs to be restricted to a sequential execution applying to the SC MoC. By detection of data dependencies and computing one possible static schedule for a program, the sequentialized SCG is generated. Thus, a sequential execution of a program is guaranteed. Hence, choosing the sequentialized SCG as starting point results in a sequential behavior of the circuit.

(b) The SSA SCG solves data-flow problems. Those problems occur if several assignments to one variable exist within one program. The SSA SCG uses SSAs as solution. Hence, all targets of multiple assignments get version numbers. In the SSA SCG the closer to the end of the program an assignment occurs the higher is the target‘s version number. This intermediate transformation was integrated as feature in KiCo and therefore improves the traceability of the circuit transformation step.

(c) After resolving the data-flow dependencies, a circuit transformation has been implemented. In this transformation assignment nodes and conditional nodes are translated into corresponding circuit components and links between those components are created.

(d) Furthermore, the circuit transformation creates an initialization region. This region contains the tick and reset logic of the circuit and provides registers for each input variable which assures the signal of inputs to be constant throughout a tick.

(e) So called *pre*-registers are provided in order to save values of output variables across the edges of ticks.

(f) To visualize the generated circuits, a KLighD circuit diagram synthesis was implemented and delivers a few options modifying the appearance of the circuit.

(g) To support the comprehensibility of the transformed circuits and going beyond the interactive incremental approach, a simulation has been implemented. This simulation adopts the information coming from the C code which is generated for the simulation of sequentialized SCGs. In each tick the active gates of the circuit are highlighted as well as all wires or links with applied voltage.

In this thesis all of these steps are conceptually introduced and their implementation is presented. Aside of supporting the comprehensibility of circuits, the simulation serves as validation tool. Used for evaluation and documented in the evaluation chapter, the simulation prooves circuits to behave as expected according to the associated sequentialized SCGs and given execution traces.

7.2 Future Work

Various works in literature cover the issue of hardware synthesis. However, this thesis delivers an interactive incremental approach of hardware synthesis exemplified for SCCharts compilation. The introduced concept is worth further enhancements. In the following a few suggestions for future work on this topic are given.

Reintroduce VHDL

The approach introduced by Johannsen [Joh13] translated SCCharts into the VHDL. With this, the usage of the code in ISE was possible. The transformation of circuits into VHDL code is desirable. Simulations and a visualization in ISE could be compared to the in KIELER created circuits and their simulation. This could signify a new resource for validation. Furthermore, FPGAs could be programed with the generated VHDL code and the program could be executed under real time conditions.

Transform Sequentially Constructive Graphs

As mentioned in Section 7.1 *(a)*, the circuit is transformed from the sequentialized SCG. On the one side, this decision assures a sequential behavior of the circuit. On the other side, this decision may be too restrictive, since the sequentialized SCG depicts solely one static schedule. The approach to transform circuits from Sequentially Constructive Program Dependency Graph (SCPDG) without sequentialization but still keeping data dependencies in mind is an interesting topic. The Sequentially Constructive Program Dependency Graph (SCPDG) was studied by Weiß [Wei15] and resolves dependencies while maintainig maximum parallelism.

Improve Validation

Section 6.3 introduces validation methods. These methods could be implemented as described and executed. Further validation methods help to guarantee the transformed circuits to be correct and behave as expected. Before FPGAs are programmed with generated VHDL code the circuits should be tested sufficiently to prevent any malfunction of the FPGAs. Moreover, sufficient testing to guarantee a smoothly run of simulations for more complex systems is a great enhancement for the incremental interactive approach.

Optimizations

Based on the synthesized hardware, several optimizations are viable. For optimization of hardware logic, laws such as , e. g., *DeMorgan's Laws* or other rules known from boolean algebra are helpful.

As seen in Figure 4.24 the ABRO circuit is relatively larger compared to the AO circuit. Figure 7.1 shows an ABRO circuit synthesized from Esterel and optimized. This circuit compared to the ABRO circuit in Figure 4.24 is remarkably smaller. The different scaling of both circuits is explainable since the Esterel ABRO has no initialization region. Furthermore, the usage of signals instead of variables simplifies the Esterel ABRO and multiple occurrences of `vcc` and `gnd` in the introduced synthesis of this thesis enlarge the generated circuits. Additionally, since the sequentialized SCGs are not optimized, the circuits transformed from them are not optimized as well. Dumitru, Edwards and Berry [PBEB07a] describe which steps lead to the optimized circuit. Part of the optimization steps should be considered to minimize sequentialized SCGs in KIELER to gain optimized circuits. The Multi-Value Logic Synthesis and Verification (MV-SIS) tool could be used for minimization as described by Gädtke [Gäd07]. The minimization further enhances efficiency in verification and understandability of the circuits.

```
module ABRO:
input A, B, R ;
output O ;
loop
  [
    await A
  ||
    await B
  ];
  emit O
each R
end module
```

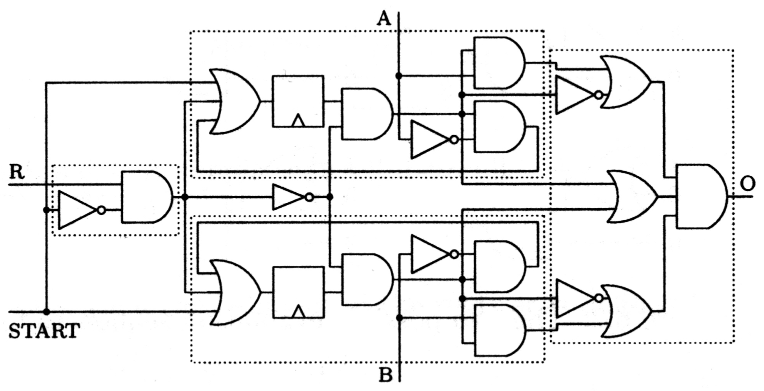


Figure 7.1. ABRO Esterel program and its circuit translation. Extracted from [PBEB07b, p. 105]

Bibliography

[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.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM. URL: http://doi.acm.org/10.1145/73560.73561, doi:10.1145/73560.73561.

[BC85] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, UK, 1985. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646723.702721.

[BDKN94] Daniel Brand, Anthony Drumm, Sandip Kundu, and Prakash Narain. Incremental synthesis. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '94, pages 14–18, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=191326.191338.

[Ber92] Gérard Berry. Mechanized reasoning and hardware design. chapter Esterel on Hardware, pages 87–104. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. URL: http://dl.acm.org/citation.cfm?id=149943.149953.

[Ber00] G. Berry. *The Esterel V5 Language Primer: Version V5_91*. Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2000. URL: https://books.google.de/books?id=JYSNMQAACAAJ.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992. URL: http://www.sciencedirect.com/science/article/pii/016764239290005V, doi:http://dx.doi.org/10.1016/0167-6423(92)90005-V.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: http://doi.acm.org/10.1145/115372.115320, doi:10.1145/115372.115320.

[DH89] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(7):798–807, Jul 1989. doi:10.1109/43.31537.

[DM99] G. De Micheli. Hardware synthesis from c/c++ models. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 382–383, 1999. doi:10.1109/DATE.1999.761150.

[Edw05] Stephen A. Edwards. The challenges of hardware synthesis from c-like languages. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 66–67, Washington, DC, USA, 2005. IEEE Computer Society. URL: http://dx.doi.org/10.1109/DATE.2005.307, doi:10.1109/DATE.2005.307.

[Gäd07] Sascha Gädtke. Hardware/Software Co-Design für einen Reaktiven Prozessor. Diploma thesis, Kiel University, Department of Computer Science, May 2007.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987. URL: http://www.sciencedirect.com/science/article/pii/0167642387900359, doi:http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991. doi:10.1109/5.97300.

[Joh13] Gunnar Johannsen. Hardwaresynthese aus SCCharts. Master thesis, Kiel University, Department of Computer Science, October 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf.

[KD90] David Ku and Giovanni DeMicheli. Hardwarec – a language for hardware design (version 2.0). Technical report, Stanford, CA, USA, 1990.

[KR00] Tommy Kuhn and Wolfgang Rosenstiel. Java based object oriented hardware specification and synthesis. In *Proceedings of the 2000 Asia* *and South Pacific Design Automation Conference*, ASP-DAC '00, pages 579–582, New York, NY, USA, 2000. ACM. URL: `http://doi.acm.org/10.1145/368434.368809`, `doi:10.1145/368434.368809`.

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

[MSF+11] Christian Motika, Miro Spönemann, Hauke Fuhrmann, Christoph Krüger, John Julian Carstens, and Reinhard von Hanxleden. KIELER Actor Oriented Modeling (KAOM). Poster presented at 9th Biennial Ptolemy Miniconference (PTCONF'11), Berkeley, CA, USA, February 2011.

[MSvH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. Compiling SCCharts—A case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8802 of *LNCS*, pages 443–462, Corfu, Greece, October 2014. `doi:10.1007/978-3-662-45234-9`.

[PAB94] S. C. Prasad, P. Anirudhan, and P. Bosshart. A system for incremental synthesis to gate-level and reoptimization following rtl design changes. In *Proceedings of the 31st Annual Design Automation Conference*, DAC '94, pages 441–446, New York, NY, USA, 1994. ACM. URL: `http://doi.acm.org/10.1145/196244.196461`, `doi:10.1145/196244.196461`.

[PBEB07a] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gerard Berry. *Compiling Esterel*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[PBEB07b] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[Ren11] Mingming Ren. *An incremental approach for hardware discrete controller synthesis*. Theses, INSA de Lyon, July 2011. URL: `https://tel.archives-ouvertes.fr/tel-00679296`.

[RSM+15] Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mendler. SCEst: Sequentially Constructive Esterel. In *Proceedings of the 13th ACM-IEEE International Conference* *on Formal Methods and Models for System Design (MEMOCODE'15)*, Austin, TX, USA, September 2015.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. URL: http://doi.acm.org/10.1145/73560.73562, doi:10.1145/73560.73562.

[SMP88] Charles E. Stroud, Ronald R. Munoz, and David A. Pierce. Behavioral model synthesis with cones. *IEEE Des. Test*, 5(3):22–30, May 1988. URL: http://dx.doi.org/10.1109/54.7960, doi:10.1109/54.7960.

[Smy13] Steven Smyth. Code generation for sequential constructiveness. Diploma thesis, Kiel University, Department of Computer Science, July 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf.

[SP98] D. Soderman and Y. Panchul. Implementing c algorithms in reconfigurable hardware using c2verilog. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 339–342, Apr 1998. doi:10.1109/FPGA.1998.707944.

[TCM06] R. Thomson, V. Chouliaras, and D. Mulvaney. The hardware synthesis of a java subset. In *Norchip Conference, 2006. 24th*, pages 217–220, Nov 2006. doi:10.1109/NORCHP.2006.329214.

[vHDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. Sccharts: Sequentially constructive statecharts for safety-critical applications: Hw/sw-synthesis for a conservative extension of synchronous statecharts. *SIGPLAN Not.*, 49(6):372–383, June 2014. URL: http://doi.acm.org/10.1145/2666356.2594310, doi:10.1145/2666356.2594310.

[vHMA+13] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2013. ISSN 2192-6247.

[vHMA+14] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation (revisited). Technical report, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2014. ISSN 2192-6247.

[Wei15] Tibor Weiß. Von Nebenläufigkeit zu Parallelität in SCCharts, October 2015.