Hardwaresynthese aus SCCharts

Gunnar Johannsen

Master Thesis
eingereicht im Jahr 2013

Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden

Betreut durch:
Dipl.-Inf. Ass. iur. Insa Marie-Ann Fuhrmann
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 Quellen und Hilfsmittel verwendet habe.

Kiel, 19.10.2013

Zusammenfassung

Sequentially Constructiveness Charts (SCCharts) ist eine grafische synchrone Modellierungssprache zum Modellieren deterministischer reaktiver Systeme. Damit Programme, die mit synchronen Sprachen erzeugt wurden, auf solchen Systemen ausgeführt werden können, werden sie zum Beispiel in C oder Java übersetzt. Ebenfalls ist die Synthese nach Hardware möglich.

Diese Masterarbeit präsentiert zwei Synthesen um aus SCCharts Hardwareschaltkreise zu synthetisieren. Vorbereitend werden SCCharts in eine sequentielle Repräsentation übersetzt [Smy13]. Ausgehend von dieser Repräsentation wird in der ersten Transformation direkt aus der sequentiellen Repräsentation sequentieller VHDL-Code generiert, der nachfolgende in Hardware transformiert werden kann. Bestehende Datenabhängigkeiten in der sequentiellen Repräsentation werden dabei vom VHDL-Compiler aufgelöst. In der zweiten Transformation werden die bestehende Datenabhängigkeiten mit Hilfe einer Transformation aufgelöst, um nachfolgend parallelen VHDL-Code zu erzeugen. Dieser kann wiederum in einen Schaltkreis synthetisiert werden.

Die vorgestellten Synthesen wurden in dem Forschungsprojekt Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) implementiert. Auf dieser Grundlage wurden die Synthesen im Rahmen dieser Arbeit evaluiert.

Weiterhin wird eine Beispielimplementierung auf einem FPGA und ein Regressionstest vorgestellt, mit dem die korrekte Funktionsweise synthetisierter Schaltkreise überprüft wurde.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

UML Unified Modeling Language

EMF Eclipse Modeling Framework

GMF Graphical Modeling Framework

MoC Model of Computation

XML Extensible Markup Language

IDE integrated design environment

KIEM KIELER Execution Manager

M2M model-to-model

RCP Rich Client Platform

IEEE IEEE

MDE Model-Driven Engineering

EDIF Electronic Design Interchange Format

SSA Static Single Assignment

VHDL Very High Speed Integrated Circuit Description Language

Verilog Verilog

ISE Integrated Software Environment

ISim ISE Simulator

Xtend Xtend

Xtext Xtext

IDE integrierte Entwicklungsumgebung

UML Unified Modeling Language

IBM International Business Machines Corporation

Tabellenverzeichnis

XMI XML Metadata Interchange

MtM Model zu Model

SCL Sequentially Constructive Language

KSbasE KIELER Structure-based Editing

KLighD KIELER Lightweight Diagrams

KIML KIELER Infrastructure for Meta Layout

KLay KIELER Layouters

UUT Unit Under Test

ISE Integrated Software Environment

RTL Register Transfer Logic

ISim ISE Simulator

IEC International Electrotechnical Commission

SCG Sequentially Constructive Graph

DCM Digital Clock Manager

CLB Configurable Logic Block

KEP Kiel Esterel Processor

Kapitel 1

Einleitung

Systeme, die kontinuierlich mit der Umwelt interagieren, nennt man reaktive Systeme, wie zum Beispiel das Motorsteuergerät eines Autos, die Airbagsteuerung oder mobile Telefone. Ein solches System durchläuft für eine Reaktion drei Phasen. In der ersten Phase werden die Eingaben der Umgebung gelesen, danach folgt die Berechnung der Reaktion auf die Eingaben, und zum Schluss werden die berechneten Ausgaben zurück an die Umgebung gegeben. Der zeitliche Ablauf reaktiver Systeme ist von der Umgebung abhängig und wird nicht durch das System selber bestimmt. Die Abbildung 1.1 zeigt ein reaktives System.

Da solche Systeme oft gleichzeitig auf mehrere Eingaben aus der Umwelt reagieren, bestehen sie aus mehreren nebenläufigen Teilsystemen. In der Regel wird für reaktive Systeme dieser Art ein deterministisches Verhalten vorausgesetzt. Solche Systeme lassen sich nur bedingt mit Sprachen wie Java oder C programmieren, da bei diesen Sprachen schwer nebenläufiger Determinismus erreicht werden kann.

So ist für die Verwendung von eingebetteten reaktiven Systemen in Bereichen der Automobil-, Flugzeug- oder medizinischen-Industrie eine deterministische nebenläufige Ausführung unerlässlich. Auftretende Fehler können gerade in diesen

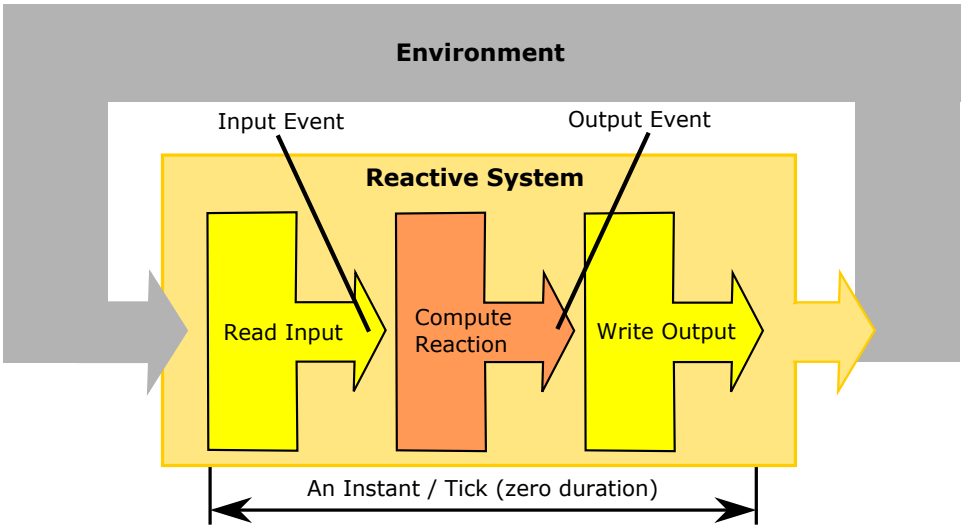


Abbildung 1.1. Ein eingebettetes reaktives System [MvHH13]

Bereichen im schlimmsten Fall Menschen das Leben kosten. Für die deterministische Programmierung von reaktiven Systemen, speziell für sicherheitskritische Systeme, wurden synchrone Sprachen entwickelt, die den Anforderungen nachkommen.

1.1 Synchrone Sprachen

Synchrone Sprachen sind speziell für die deterministische nebenläufige Programmierung von reaktiven Systemen entwickelt. Sie unterbinden nichtdeterministische Ausführungen nebenläufiger Threads, im Gegensatz zu C oder Java, welches einer der Hauptgründe für deren Nichtdeterminismus ist. Die synchronen Sprachen stellen Konstrukte zur Programmierung von deterministischer Nebenläufigkeit zur Verfügung. Bei synchronen Sprachen wird von der Zeit abstrahiert, was eine Trennung zwischen der eigentlichen Funktion und des zeitlichen Verhaltens zulässt. Zu den synchronen Sprachen gehören unter anderem Esterel [BC84], Lustre [HCRP91], SCADE [Est08], SyncCharts [And96] und SCCharts [vHMA+13a].

Synchrone Sprachen basieren auf synchronen Ausführungsmodellen, die die deterministische Ausführung von Programmen sicherstellen. Dem synchronen Ausführungsmodell liegt die Synchronizitätshypothese zugrunde, die folgende Eigenschaften beschreibt [PBdST05]:

- Perfekte Synchronizität

  Ein System läuft in perfekter Synchronizität, wenn alle Prozesse gleichzeitig ablaufen und für die Berechnungen keine Zeit verbraucht wird. Das bedeutet, dass in dem selben Moment, in dem Eingaben gelesen werden, die Ausgaben berechnet und ausgegeben werden.

- Zero-delay

  Zero-delay besagt, dass bei einem Zustandswechsel innerhalb eines Programms, von einem Zustand in seinen Folgezustand, keine Zeit verbraucht wird.

- Multiform notion of time

  Bei der *Mutiform notion of time* wird von der physikalischen Zeit abstrahiert. Die Zeit wird durch eine Reihenfolge von Ereignissen bestimmt. Soll die Zeit durch die physikalische Zeit modelliert werden, können periodische Ereignisse gewählt werden, die zum Beispiel jede Sekunde auftreten.

Im synchronen Ausführungsmodell wird die Zeit in diskrete *Ticks* eingeteilt. Dabei werden zu Beginn des Ticks die Eingaben gelesen und am Ende eines Tick die Ausgaben produziert. Ein solcher Tick heißt *Makrotick* oder einfach *Tick*. Einem solchen Tick wird zunächst keine Ausführungszeit zugeschrieben, siehe Abbildung 1.1

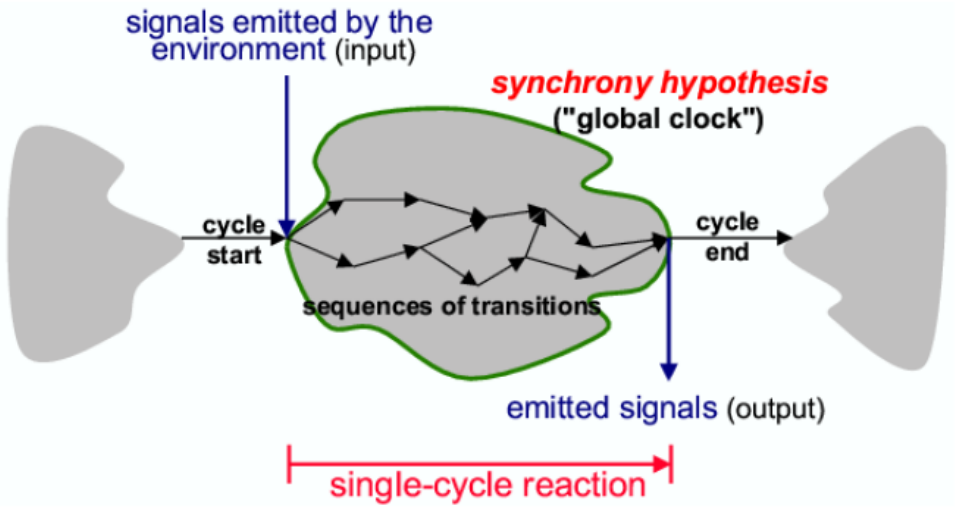


Abbildung 1.2. Die Synchronizitätsshypothese (G. Luettgen, 2001)

(*zero duration*). Die Berechnung kann in einzelne Berechnungsschritte, sogenannte *Mikroticks* aufgeteilt, in denen die Ausgaben berechnet werden. In Abbildung 1.2 ist die Berechnung eines Ticks dargestellt.

1.1.1 SyncCharts

SyncCharts basieren auf den *Statecharts* von David Harel [Har87] und erweitern diese um die synchronen Eigenschaften wie Determinismus. Bei Statecharts handelt es sich im Grunde um hierarchische Mealy-Automaten mit Preemption und Kommunikationsmechanismen für nebenläufige Threads zur nebenläufigen Modellierung. Diese sind jedoch nicht deterministisch. SyncCharts hingegen basieren, wie Esterel, auf dem synchronen Ausführungsmodell von Berry, welches den Determinismus sicherstellt, und auf der Synchronizitätshypothese. *SyncCharts* ist eine grafische Modellierungssprache für synchrone Sprachen. Sie wurde 1996 von Charles André [And96] als ein grafisches Pendant von Esterel [BC84] vorgestellt. Um das Verständnis von SynchCharts zu erleichtern, ist in Abbildung 1.3 ein SyncChart gezeigt, welches nun näher erklärt werden soll.

Es handelt sich hierbei um das “Hello World” der synchronen Sprachen, das ABRO-Beispiel. ABRO verwendet die typischen Modellierungskomponenten von SyncCharts.

▻ Zustände

Bei Zuständen kann zwischen einfachen und *Makrozuständen* unterschieden werden. Einfache Zustände sind zum Beispiel der Zustand done in der Abbildung 1.3. Makrozustände beinhalten weitere SyncCharts Elemente, wie zum Beispiel der

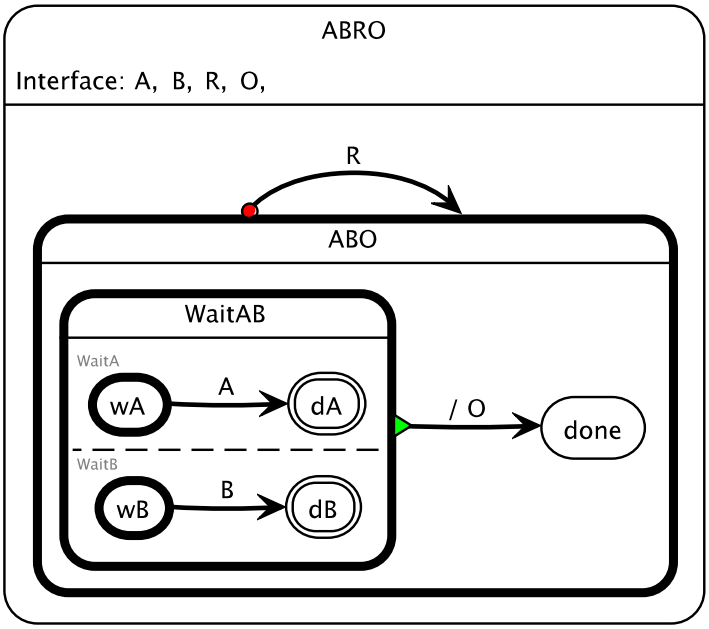


Abbildung 1.3. Das “Hello World” der SyncCharts — ABRO [Smy13]

Zustand WaitAB. Es können eine oder mehrere parallele Regionen in einem Makrozustand enthalten sein (siehe Zustand WaitAB). Ein Zustand kann optional als initialer Zustand (siehe Zustand wA) markiert sein (dicker Rand) oder als Endzustand (siehe Zustand dA) (doppelter Rand).

▻ Transition

Transitionen werden verwendet, um von einem Zustand in einen Folgezustand zu wechseln, dabei können drei Transitionstypen unterschieden werden. *Weak Transitions* werden zum einfachen Zustandswechsel verwendet. *Normal termination Transitions* werden verwendet, um nebenläufige Regionen zusammenzuführen. Die normal termination Transition wird genommen, wenn alle Regionen terminiert sind. Sie werden durch einen grünen Pfeil gekennzeichnet. *Strong abort Transitions* unterbrechen sofort die Ausführung des Makrozustands und setzen die Ausführung an ihrem Zielzustand fort (Preemption). Sie sind mit einem roten Punkt gekennzeichnet. Transitionen können einen Trigger und ein Effekt besitzen, beide sind optional. Eine Transition kann nur genommen werden, wenn in einem Tick der Zustand aktiv ist von dem sie ausgeht und wenn der Trigger wahr ist. Damit sie genommen wird, muss sie weiterhin die höchste Priorität haben und darf nicht preemted werden. Prioritäten entscheiden, welche Transition genommen wird, wenn ein aktiver Zustand mehr als eine ausgehenden Transition besitzt die

wahr ist. Der Effekt wird ausgeführt, wenn die Transition genommen wird. So wird zum Beispiel die Transition vom Zustand wA zum Zustand dA genommen, wenn ein A registriert wurde. Ein O wird emittiert wenn die normal termination Transition von Zustand WaitAB genommen wird.

▻ Signale

In SyncCharts werden Signale zur Kommunikation verwendet. Signale werden im äußersten Zustand, dem *Main State*, im Interface deklariert. Signale in SyncCharts unterliegen dem *Signalkohärenz Gesetz*, siehe Unterkapitel 1.1.1. Sie haben einen Signalstatus der in einem Tick entweder present oder absent ist (*pure Signals*), wertbehaftete Signale (*valued Signals*) haben darüber hinaus noch einen zusätzlichen Wert, der als Zustand über Tickgrenzen hinweg erhalten bleibt. Der Status eines Signals wird auf present gesetzt, wenn es emittiert wird. Wertbehaftete Signale besitzen einen Typ, zum Beispiel Integer oder Boolean. Signale können als Ein- und Ausgangssignale definiert werden, um mit der Umgebung zu interagieren. Ein Signal kann auch ein lokales Signal sein, diese können im jeweiligen Zustand deklariert werden.

▻ Regionen

Regionen dienen zu Spezifikation von Nebenläufigkeit in SyncCharts. Sie müssen einen initialen Zustand besitzen. Regionen können Endzustände besitzen. Ist ein Endzustand einer Region aktiv, ist die Ausführung der Region beendet und sie terminiert.

In der Abbildung 1.3 sind einfache Zustände und hierarchische Zustände mit nebenläufigen Regionen zu sehen. Nachdem das SyncChart gestartet wurde, sind die beiden initialen Zustände wA und wB sowie WaitAB und ABO aktiv. Der Zustand WaitAB terminiert, wenn die beiden internen Regionen terminiert sind. Die beiden Regionen terminieren, wenn jeweils die Eingaben A beziehungsweise B registriert wurden. Wenn die Region WaitAB terminiert, wird die ausgehende Transition von WaitAB genommen und O emittiert. Im Zustand done bleibt die Ausführung stehen. ABO kann erneut gestartet werden, wenn das Signal R present ist. Das SyncChart kann zu jedem Zeitpunkt, unabhängig davon, in welchem Zustand sich das SyncChart befindet, mit dem Signal R neu gestartet werden.

Die Sprachen Esterel und SyncCharts basieren auf dem synchronen Ausführungsmodell von Berry [Ber02]. Die Berry Konstruktivität legt die Synchronizitätsshypothese und das Signalkohärenz Gesetz zugrunde. Das Signalkohärenz Gesetzt definiert, welche Status Signale in einem Tick besitzen dürfen:

Signalkohärenz Gesetz

Ein Signal, ist in einem Tick nur present (vorhanden) und nur dann, wenn es in diesem Tick emittiert worden ist. Ein Signal ist im Normalfall absent (nicht vorhanden). Weiterhin kann ein Signal niemals in einem Tick beide Zustände annehmen.

1.1.2 Sequentiell konstruktives Ausführungsmodell

Das synchrone Ausführungsmodell von Berry stellt Determinismus in einer nebenläufigen Programmierung zur Verfügung, jedoch mit einigen Einschränkungen. Es ist in der Berry Konstruktivität aufgrund des Signalkohärenz Gesetzes verboten, dass Signale in einem Tick mehrere unterschiedliche Status besitzen.

Das Signalkohärenz Gesetz beschränkt die Klasse der gültigen Programme. Eine Auflockerung des Gesetzes würde die Klasse gültiger Programme vergrößern. Durch diese Änderung müssten, um den Determinismus zu erhalten, weitere Ausführungseigenschaften eingehalten werden. Das *Sequentiell Konstruktive Ausführungsmodell*, vorgestellt in “Sequentially Constructive Concurrency — A conservative extension to the synchronous model of computation” [vHMA$^+$13b] von von Hanxleden et al., löst die Einschränkungen der Berry Konstruktivität auf. In diesem Ausführungsmodell werden Variablen anstelle von Signalen für die Kommunikation verwendet. Variablen können mehrere Werte in einem Tick führen, solange für das Programm mindestens ein sequentielles statisches Schedule existiert und alle existierenden Schedules das gleiche Ausgabeverhalten nach sich ziehen. Es handelt sich dabei um eine Erweiterung der klassischen Berry Konstruktivität. Es werden weiterhin Signale unterstützt, so dass SyncCharts mit diesem Ausführungsmodell ebenfalls ausgeführt werden können. Eine deterministische Ausführung von Programmen ist durch das sequentiell konstruktive Ausführungsmodell weiterhin gewährleistet.

1.1.3 SCCharts

Sequentially Constructive Charts (SCCharts) [vHMA$^+$13a] stellen eine grafische Modellierungssprache dar, die auf dem sequentiell konstruktiven Ausführungsmodell beruht. Im Gegensatz zu SyncCharts werden in einem SCChart Variablen verwendet. Variablen können mehrere Werte in einem Tick besitzen. Ein SCChart ist konstruktiv, wenn es syntaktisch korrekt ist und mindestens ein sequentielles Schedule existiert. Alle existierenden Schedules müssen in jedem Tick das selbe deterministische Verhalten aufweisen.

SCCharts bestehen, wie SyncCharts, aus Zuständen, Transitionen und Regionen. Es gibt eine minimale Anzahl an Modellierungskomponenten, die sogenannten *Core SCCharts*, mit denen vollständige SCCharts modelliert werden können. In einer erweiterten Version, den *extended SCCharts*, stehen syntaktisch angereicherte Elemente zur Verfügung, mit denen komplexere Sachverhalte einfacher modelliert werden können. Eine detailliertere Erklärung der SCCharts wird in Abschnitt 4.1.2 gegeben.

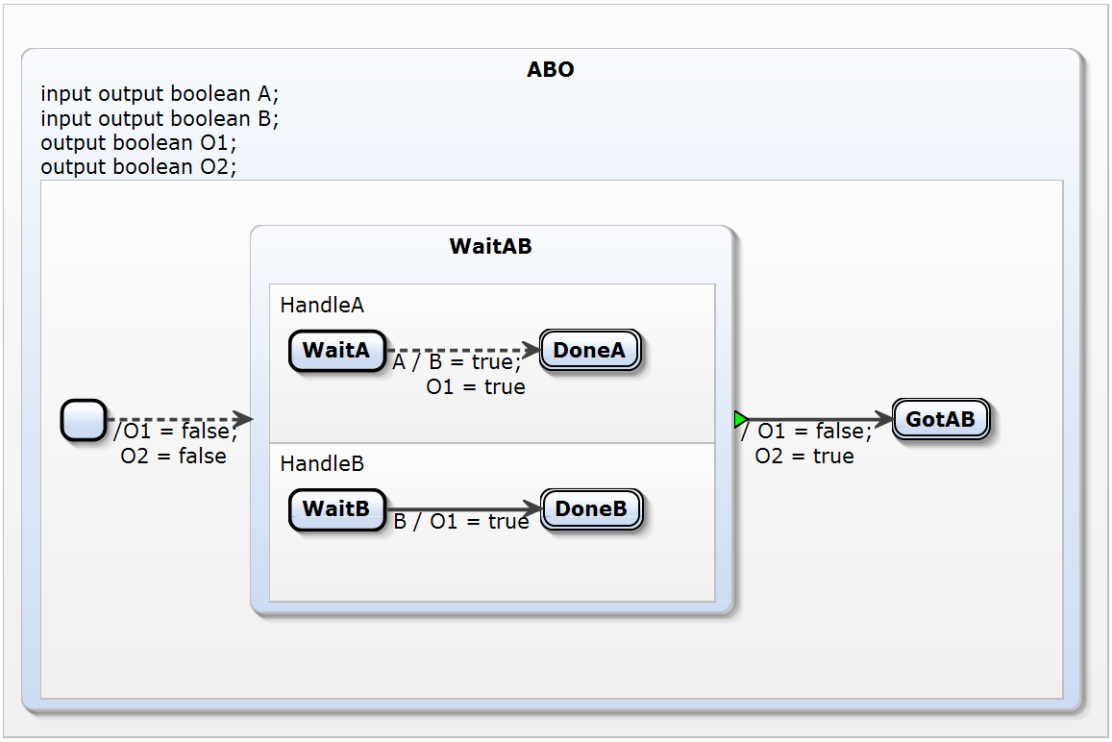


Abbildung 1.4. Das ABO Beispiel modelliert als Core SCChart

In Abbildung 1.4 ist ein Core SCCharts Beispiel gezeigt. Es werden in SCCharts ähnliche Modellierungskomponenten verwendet wie in SyncCharts. Eine genaue Erklärung zum Beispiel ist in Abschnitt 4.1.2 zu finden.

1.2 Aufgabenstellung

Die Programmierung eingebetteter reaktiver Systeme ist, aufgrund der Anforderung, dass trotz Nebenläufigkeit der Determinismus garantiert sein muss, eine Herausforderung. Sprachen wie Esterel oder SCCharts sind dem Programmierer behilflich, Systeme mit solchen Anforderungen zu entwickeln. Die entwickelten Programme können zum Beispiel mit Hilfe von Transformationen von synchronen Sprachen zu C oder Java auf Mikrocontroller programmiert werden. Bei reaktiven Systemen muss es sich jedoch nicht nur um Mikrocontroller gesteuerte Systeme handeln. Gerade, wenn schnelle oder massiv parallele Anforderungen an ein reaktives System gestellt werden, kommen FPGAs zu Einsatz. An die Entwicklung hochintegrierter paralleler Schaltkreise für FPGAs, werden spezifische Anforderungen gestellt. Ein Problem besteht darin, dass gerade in der VHDL Programmierung “echte” Parallelität vorherrscht und sequentielle Ausführungen abgebildet werden müssen. Ebenso ist die deterministische Programmierung von Schaltkreisen eine sehr komplexe Aufgabe, da

neben der allgemeinen Schwierigkeit der parallelen Programmierung, noch zeitliche Aspekte wie Signallaufzeiten auftreten.

Für die Hardwaresynthese von SCCharts gibt noch keine direkte Transformation. In dieser Arbeit soll eine Lösung präsentiert werden, mit der SCCharts in einen Hardware Schaltkreis übersetzt werden können, der das selbe deterministische Verhalten aufweist wie das SCChart. SCCharts werden in einem ersten Schritt, mit den von Steven Smyth entwickelten Transformationen [Smy13], in ein sequentielles Programm übersetzt. In der Hardwareprogrammierung handelt es sich jedoch überwiegend um eine parallel Programmierweise. Damit die sequentielle Ausführung des Programms in Hardware umgesetzt werden kann, müssen die Datenabhängigkeiten zwischen einzelnen sequentiellen Anweisungen aufgelöst werden. Des Weiteren ist eine gesonderte Betrachtung der Ein- und Ausgabevariablen eines SCCharts bei der Hardwaresynthese notwendig. Für Ein- und Ausgabewerte müssen Mechanismen entwickelt werden, damit VHDL-Signale zum richtigen Zeitpunkt gelesen und geschrieben werden. Da Signalwerte bei Schaltkreisen flüchtig sind, muss eine Möglichkeit geschaffen werden, den Zustand, in dem sich ein SCChart befindet, in Hardware persistent abzubilden. Ebenso werden Techniken benötigt, die die Berechnungen des Schaltkreises starten und gegebenenfalls auch zurücksetzen. Diese Arbeit stellt für die vorgestellten Problemstellungen Lösungen bereit, um somit SCCharts auf Hardware auszuführen zu können. Da die Validierung von Schaltkreisen äußerst komplex ist, müssen Möglichkeiten geschaffen werden, diese zu validieren. Die Arbeit stellt einen Regressionstest vor, mit dem automatisch eine beliebige Anzahl an SCCharts-Transformationen getestet werden kann, um eine korrekte Transformation von SCCharts zu Hardware sicherzustellen.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit, die im weiteren Verlauf vorgestellt wird, ist wie folgt organisiert:

In Kapitel 2 werden verschiedene Konzepte der Codegenerierung vorgestellt. Das Hauptaugenmerk liegt dabei auf Transformationen von sequentiellen textuellen Sprachen, wie C oder Java, und synchronen Sprachen, wie Esterel und SyncCharts, zu einer Hardwarebeschreibungssprache. Die Konzepte werden vorgestellt und Unterschiede zur vorliegenden Arbeit genannt.

Kapitel 3 beschreibt die in dieser Arbeit verwendeten Technologien. So wird das auf Eclipse basierende Projekt KIELER vorgestellt und die Technologien Xtend, JUnit und VHDL eingeführt. Ebenso werden das Entwicklungswerkzeug ISE und die Simulationsumgebung ISim präsentiert.

Die Konzepte und Ideen zweier möglicher Lösungsansätze werden in Kapitel 4 präsentiert. Dazu werden zu Beginn das sequentiell konstruktive Ausführungsmodell sowie die Sprachen SCChart und SCL vorgestellt. Nachfolgend werden zwei Ansätze

für die Transformation von SCCharts zu VHDL präsentiert.

In Kapitel 6 werden Details zur Implementierung in dem KIELER-Projekt vorgestellt. Dies umfasst die Organisation der benötigten Transformationen in die verschieden Java und Xtend basierten Plugins in Eclipse, sowie eine allgemeine Beschreibung der Transformationen.

Eine automatische Validierung der Transformationsergebnisse wird in Kapitel 5 vorgestellt. Nachdem ein Format zum Testen von SCCharts dargelegt wurde, wird eine Möglichkeit zum Testen synthetisierter Schaltkreise mit dem Xilinx Tool ISim erläutert. Nach der Vermittlung der Idee des Tests, wird ein automatischer Test zum Überprüfen mehrerer Modelle präsentiert.

Ein Vergleich der beiden in Kapitel 4 vorgestellten Lösungsansätze wird in Kapitel 7 durchgeführt. Dort werden neben der Chipauslastung des FPGAs auch die Ausführungsgeschwindigkeiten der beiden Ansätze miteinander verglichen.

Im letzten Kapitel, Kapitel 8, wird die Arbeit in einer kurzen Zusammenfassung rekapituliert und weiterführende Ideen werden vorgestellt.

Kapitel 2

Verwandte Arbeiten

Die Synthese von Hardware aus den unterschiedlichsten Programmiersprachen ist ein beliebtes Forschungsumfeld. Die Verwendung bekannter Programmiersprachen oder abstrakter Modellierungssprachen zur Synthese von Hardware, erleichtert es Programmierern aus dem Softwareumfeld Schaltkreise zu entwickeln. In diesem Abschnitt der Arbeit werden Sprachen, die sich für die Hardwaresynthese eignen, vorgestellt und mit dem hiesigen Ansatz verglichen.

2.1 Softwaresynthese aus SyncCharts

Der Statechart Dialekt SyncCharts ist speziell für die Entwicklung und Programmierung deterministischer reaktiver eingebetteter Systeme entwickelt worden. Bei eingebetteten Systemen handelt es sich meistens um Systeme die von einem Mikrocontroller oder einer sonstigen Hardware gesteuert werden. Sie werden in der Regel mit der Programmiersprache C programmiert. Die grafische Repräsentation von SyncCharts muss zuvor in ein äquivalentes C Programm transformiert werden, bevor der Controller das Programm ausführen kann. Dabei müssen deterministische Nebenläufigkeit und Preemption erhalten bleiben. Von Hanxleden stellt *SyncCharts in C (SC)* [vH09] vor, um diese Eigenschaften zu erhalten. Ein leichtgewichtiger Ansatz mit dem C Programme mit Determinismus, Nebenläufigkeit und Preemption realisiert werden können. SC besteht dabei aus einer kleinen Anzahl an C Makros.

Traulsen, Amende und von Hanxleden stellen einen Weg vor, SyncCharts zu SC zu transformieren [TAvH11]. Der Vorteil der Transformation spiegelt sich in der Lesbarkeit und in der Strukturerhaltung zwischen dem SyncChart und dem C Code wider.

Ebenso können SyncCharts in Java abgebildet werden. Da mit Java nur schwer deterministische Nebenläufigkeit realisiert werden kann, stellen Motika, von Hanxleden und Heinold *Synchronous Java (SJ)* [MvHH13] vor, mit dem deterministische Nebenläufigkeit umgesetzt werden kann. SyncCharts können somit in SJ realisiert werden.

SC und SJ stellen die Möglichkeit bereit, mit den Programmiersprachen C und Java, ein deterministisches Verhalten zu programmieren, sie sind dabei keine eigenständigen Sprachen. Damit die vorgestellten Ansätze für die Hardwaresynthese verwendet werden könnten, müssen SyncCharts in einer weiteren Transformation

von C bzw. Java in Hardware übersetzt werden. Es gibt Übersetzungen von Sprachen wie C oder Java zu Hardware. Einige Ansätze werden in den folgenden Kapiteln präsentiert.

2.2 Hardwaresynthese

Die Herstellung hochintegrierter Schaltkreise ist aus der heutigen Elektronikbranche nicht wegzudenken. Früher wurden Schaltungen von Hand erstellt. Mit der Entwicklung von Hardwarebeschreibungssprachen wie VHDL und Verilog können Schaltungen in Software erstellt, simuliert und getestet werden. Ein großer Vorteil dabei ist die Verkürzung von Entwicklungs- und Produktionszyklen.

Die Programmierung schneller und komplexer Schaltungen für Hardware birgt besondere zeitliche Anforderungen. Bei der herkömmlichen Programmierung muss in der Regel das Zeitverhalten eines Programms nicht berücksichtigt werden und Nebenläufigkeiten werden in Form von Threads programmiert. In der Hardwaresynthese hingegen wird der Quellcode parallel ausgeführt. Um Programmierern den Einstieg und die Programmierung in der Hardwarebeschreibung zu erleichtern oder ihn sogar zu umgehen, wird an verschiedenen Möglichkeiten geforscht, C, Java und andere Programmiersprachen zu synthetisieren. Des Weiteren werden Beschreibungssprachen entwickelt, die sich wie C programmieren lassen, jedoch direkt in Hardware übersetzt werden können. Die Forschung ist ebenfalls interessant für Hardware-Software-Co-Design [DMG97]. Im folgenden Verlauf werden einige Ansätze zur Hardwaresynthese aus den Sprachen C und Java vorgestellt.

2.2.1 Hardwaresynthese aus den Sprachen C und Java

Die Programmiersprache C ist unter anderem dafür ausgelegt, Code für Ein- und Mehrkernprozessoren zu programmieren. C ist nicht dafür entwickelt worden, um Schaltkreise zu erstellen. In Hardware werden strukturelle Informationen wie Pin/Port-Beschreibungen benötigt. Ebenso ist die Hardwareprogrammierung eine überwiegend parallele Programmierweise, welche C nicht direkt zur Verfügung stellt. Weiterhin fehlen in solchen Sprachen Konstrukte, um Zeitverhalten abbilden zu können, wie zum Beispiel das einer Variablen, nach einer Zeit t, ein Wert zugewiesen wird. De Michile beschreibt in seinem Artikel “Hardware Synthesis from C/C++ Models” [DM99] solche fehlenden Eigenschaften und Probleme.

Stephen A. Edwards stellt in seinem Artikel “The Challenges of Hardware Synthesis from C-like Languages” [Edw05] einige Sprachen vor, welche *C* sehr ähnlich sind, sich jedoch zur Hardwarebeschreibung eignen. Stroud et al. präsentieren die Sprache *Cones* [SMP88], die jede C-Funktion in einen kombinatorischen Block in Hardware übersetzt. Cones implementiert unter anderem bedingte Sprünge, Schleifen und Arrays.

HardwareC [KLM90] ist eine Beschreibungssprache für das Verhalten von Hardware. Es können strukturelle und hierarchische Beschreibungen vorgenommen werden. HardwareC basiert auf der Programmiersprache C und wurde unter anderem, um Beschreibungen für nebenläufige Prozesse, *Message Passing* und das explizite Erstellen von Prozeduren, erweitert.

C2Verilog [SP98b, SP98a] stellt Transformationen von C zu *Verilog* bereit. Es wird ein Großteil des *ANSI C*-Standards unterstützt. Beispielsweise werden Pointer, Rekursionen, dynamische Speicherallozierung und weitere C-Konstrukte transformiert.

Java ist heutzutage eine der am weit verbreitetsten Programmiersprachen und es liegt nahe, ebenfalls Hardwaresynthese aus Java zu betreiben.

T. Kuhn und W. Rosenstiel [KR00] verwenden die objektorientierte Sprache Java, mit der Erweiterung *JavaBeans*. Sie analysieren iterativ alle Threads in einem Java Quellcode. Jeder Thread wird in einem Kontrollflussgraphen abgebildet, der wiederum in einen VHDL-Prozess transformiert wird. Die generierten VHDL-Prozesse können anschließend synthetisiert werden.

Cardoso und Neto [CN99] analysieren hingegen Java-Byte-Code und generieren Quellcode für Hardware-Software-Co-Design. Aus dem Byte-Code heraus werden alle Abhängigkeiten innerhalb des Quellcodes und die parallelen Programmregionen ermittelt. Programmteile, die in Hardware ausgeführt werden sollen, werden synthetisiert und auf einer rekonfigurierbaren Hardware ausgeführt.

Die Transformation von herkömmlichen Programmiersprachen zu Hardware wurde hier nicht verwendet, da bei der Transformation zur Hardware Probleme wie Zeitverhalten und Parallelität gesondert betrachtet werden müssen [DM99]. Weiterhin gehen über diesen Weg, SCCharts zum Beispiel nach SC und anschließend mit einer der vorgestellten Möglichkeiten zu synthetisieren, die strukturelle Ähnlichkeit verloren, was eine Validierung zusätzlich erschwert.

2.2.2 Esterel

Esterel [Ber00, BC84] ist eine synchrone imperative Programmiersprache. Sie ist dafür konzipiert, reaktive Systeme mit deterministischem Verhalten zu programmieren. Entwickelt wurde sie, um kontrolldominierte Software bzw. Hardware zu entwerfen. Esterel basiert auf einem streng formalen Modell und auf der synchronen Hypothese [BC84]. Das synchrone Ausführungsmodell zerlegt die Zeit in diskrete Schritte, die sogenannten *Ticks*. Das zeitliche Verhalten und die Funktionalität können so voneinander getrennt werden. Die Esterel Befehle lassen sich in zwei Gruppen einteilen. Es gibt instantane und verzögerte Befehle. Verzögerte Befehle wie zum Beispiel pause beschreiben Tickgrenzen. Ein *Tick* endet an solchen Befehlen. Während loop, emit (setzt ein Signal) oder present (testet, ob ein Signal vorhanden ist) instantane Befehle sind. Esterel-Programme werden durch Eingaben gesteuert. Ein Esterel-Programm wartet somit auf Eingabesignale und berechnet daraufhin die

zugehörigen Ausgabesignale. Der folgende Text soll eine Idee darüber vermitteln, wie Schaltkreise aus Esterel-Programmen transformiert werden können.

Schaltkreissemantik

Esterel ist so konzipiert, dass aus einem Esterel-Programm sowohl Software als auch Hardware generiert werden kann. Für die Esterel-Semantik gibt es mehrere Formalisierungen [BC84]. Für die Hardwareentwicklung ist die Schaltkreissemantik von Bedeutung. Die Transformation ist strukturell, d.h. für jeden Basisbefehl wird zunächst ein äquivalenter Schaltkreis erzeugt. Die einzelnen Schaltkreise werden mit Hilfe von zusätzlichen Gattern und Schaltdrähten, siehe Abbildung 2.1, zu einem funktionierenden Schaltkreis verbunden [Ber02], der das Verhalten des Programms in Hardware widerspiegelt.

Jedes beliebige Esterel-Programm P, ein Beispiel ist in Listing 2.1 gezeigt, wird in einen Programmblock übersetzt. Ein Block besteht aus der folgenden Schnittstellenbeschreibung: E sind Eingaben von der Umgebung, E' sind Ausgaben an die Umgebung, RES setzt die Ausführung des Blocks fort, SUSP suspendiert die Ausführung des Blocks, KILL setzt die internen Register auf Grund eines Abbruchs zurück und SEL signalisiert, dass ein Block auf eine Weiterausführung wartet (zum Beispiel pause). Der Eingang GO startet die Ausführung eines Blockes und Kn, $n \in \mathbb{N}$ signalisieren, wie der Block beendet wurde, der sogenannte *Completions Code*. Die verschiedenen Completions Code zeigen an, wie der Block terminiert, K0: der Block terminiert, K1: eine pause terminiert, Kn mit n>1: die Ausführung des Blockes wurde abgebrochen.

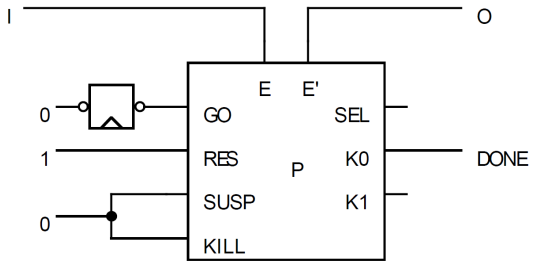


Abbildung 2.1. Allgemeiner Esterel Hardware Block [Ber02]

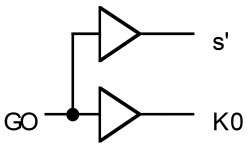
```
1  module MAIN:
2
3  input I:
4  output O:
5
6  P
7
8  end
```

Listing 2.1. Esterel Software Modul

Die Abbildung 2.2 zeigt, wie ein Signal in Esterel emittiert wird. Das GO Signal startet die Ausführung und setzt das Signal s' auf true, es wird emittiert. Da der emit Befehl instantan ist, wird der *Completion Code* K0 ebenfalls gesetzt. Abbildung 2.3 zeigt die Umsetzung eines pause Befehls in Hardware. Das Besondere an einem pause Befehl ist das Register, welches nach Erreichen der pause gesetzt wird, damit die Ausführung im nächsten *Tick* entsprechend fortgesetzt wird. Die Befehle eines

```
1 emit S;
```

(a) Esterels emit Befehl in Software

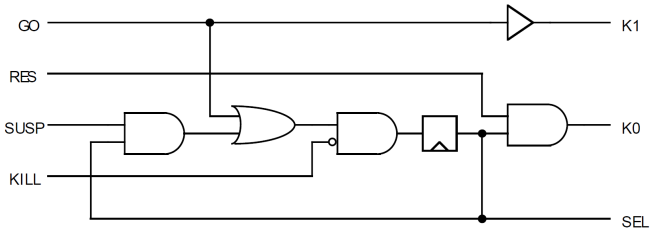


(b) Esterels emit Befehl in Hardware [Ber02]

Abbildung 2.2. Der Esterel emit Befehl in Soft- und Hardware

```
1 pause;
```

(a) Esterels pause Befehl in Software



(b) Esterels pause Befehl in Hardware [Ber02]

Abbildung 2.3. Der Esterel pause Befehl in Soft- und Hardware

Esterel-Programms werden in die einzelnen Schaltungen übersetzt und im nächsten Schritt zu einem vollständigen Schaltkreis verbunden.

Mit Esterels Schaltkreissemantik ist es möglich, konstruktive Esterel-Programme in verzögerungsunabhängige Hardware zu übersetzen [Ber00]. Mit Esterel können SyncCharts abgebildet werden. Da SCCharts auf einem anderem Ausführungsmodell beruhen, wurde die Schaltkreissemantik nicht für die Synthese von SCCharts verwendet. Die Idee, dass für pause-Anweisungen Register verwendet werden, um Tickgrenzen zu beschreiben, wurde allerdings in dieser Transformation verwendet.

2.2.3 Hardwaresynthese aus Statecharts

An der Hardwaresynthese von *Statecharts* [Har87] wurde ebenfalls geforscht. D. Harel und D. Drusinsky verwenden Statecharts zur Hardwarebeschreibung und zur Synthese von digitalen Schaltungen [DH89]. Sie verwenden dabei einzelne Automaten, die in Hardware umgesetzt werden. Zusammengeschlossen in einem Automaten-Baum bilden die einzelnen Automaten das Verhalten des Statecharts ab. Da Statecharts ein nichtdeterministisches Verhalten besitzen, eignet sich diese Synthese nicht für die Transformation von SCCharts.

2.3 Kiel Esterel Processor

Die Modellierungssprache Esterel kann nach Software oder Hardware übersetzt und ausgeführt werden. Dabei handelt es sich um zwei herkömmliche Ansätze. Ein weiterer Ansatz ist es, Esterel auf einem spezialisierten Prozessor, einem reaktiven Prozessor, auszuführen. Xi und von Hanxleden stellen den reaktiven Kiel Esterel Processor (KEP) [LvH06, LvH05] vor, dessen Befehlssatz eng an Esterel angelehnt ist. Basierend auf dem Prozessor hat sich Starke in seiner Diplomarbeit [Sta09] damit auseinander gesetzt, *Safe State Machines* (SyncCharts) auf diesem Prozessor auszuführen.

Für den KEP können nur eine bestimmte Teilmenge an zykelfreien Esterel Programmen verwendet werden [Gäd07]. Da der Prozessor an die Esterelbefehle angelehnt ist, können Esterel Befehle selbst direkt auf dem Prozessor ausgeführt werden. Die Auswertung komplexer Ausdrücke stellt sich als aufwändiger heraus. Gädtke, Traulsen und von Hanxleden präsentieren einen HW/SW Co-Design Ansatz [GTvH07] in dem komplexe Ausdrücke in Hardware transformiert und zur Ausführung ausgelagert werden.

Bei dem KEP handelt es sich um eine spezialisierte Hardware, die stark an Esterel angelehnt ist. Um eine möglichst große Flexibilität der späteren Hardware zu wahren, hat sich dieser Ansatz nicht als eine Möglichkeit herausgestellt. Ebenso können aufgrund dessen, dass die Berry Semantik im KEP zugrunde liegt, Variablen in SCCharts nicht mehrere Werte in einem Tick besitzen, wenn sie ausgeführt werden.

2.4 Das Electronic Design Interchange Format

Das Electronic Design Interchange Format (EDIF)[1] ist ein standardisiertes, maschinenlesbares Format, welches zum Austausch von Netzlisten und Schaltplänen verwendet wird. Es wurde zuletzt in der Version 4 0 0 im August 1996 durch die International Electrotechnical Commission (IEC) standardisiert. 1980 besaß die Industrie eine Reihe von Formaten, in denen die Unternehmen ihre Daten vorhielten. Das EDIF-Format wurde als neutrales Format entwickelt, um die Anzahl der Konvertierungen der Daten, die zum Austausch nötig waren, zu minimieren. Das EDIF-Format hat bis heute Bestand und wird zum Beispiel von der Firma Xilinx unter anderem zur Synthese [Inc12a] und zur Analyse des Stromverbrauchs synthetisierter Schaltkreise [SKB02] verwendet.

Die Informationen, die EDIF bereitstellt, können in zwei Gruppen eingeteilt werden [LK93]: allgemeine Design-Informationen und Informationen zur Repräsentation einer Zelle. Die Design-Informationen stellen Daten zur hierarchischen Sicht, zur grafischen Repräsentation, zur Verbindung von Einzelkomponenten und zum Hersteller/Eigentümer bereit. In den Zell-Informationen werden Verbindungsinformationen und Informationen zum Zeitverhalten gespeichert, sowie Daten zum Schaltplan

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

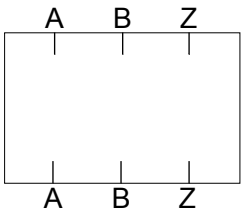
```
(Cell NAND2
  (View Topological Logic
    (Interface
      (Declare (INPUT PORT a b) (OUTPUT PORT z))
      (Body
       (Figure Group SchematicSymbol
          (shape (20 0) (arc (40 0) (60 20) (40 40)) (20 40))
          (circle (60 2]) (70 20))
         )
       )
      (PortImplementation a (Figure Group SchematicPort
                                  (Path (0 30) (20 30))))
      (PortImplementation b (Figure Group SchematicPort
                                  (Path (0 10) (20 10))))
      (PortImplementation z (Figure Group SchematicPort
                                  (Path (70 20) (90 20))))
      (Permutable a b)
    )

    (Contents
      (Instance Nand2Primitive)
    )
  )
 (View Physical
    (Interface
      (Declare (INPUT PORT a b) (OUTPUT PORT z))
      (Body (Figure Group MetalBlockage (Rectangle .... )))
      (PortImplementation b (Figure Group Poly . . . . . ))
      . . .
      (Permutable a b)
   )

   (Contents
      (Figure Group Metal (Path ...) (Polygon ...) ...)
      (Figure Group Poly (Path ...) (Path ...) ...)
      . . .
    )
  )
)
```

(a) EDIF Codeausschnitt

(b) Logische Sicht



(c) Physikalische Sicht

Abbildung 2.4. EDIF Codeausschnitt eines NAND-Gatters inkl. der logischen und physikalischen Sicht des Gatters [Cra84]

selbst. Mittels EDIF können somit elektronische Schaltungen abgebildet, gespeichert und ausgetauscht werden. Aus dieser Beschreibung kann nachfolgend Hardware synthetisiert werden.

Abbildung 2.4 zeigt einen Ausschnitt aus einer EDIF-Beschreibung eines *NAND2*-Gatters, ebenso wie die logische und physikalische Repräsentation. Auf die genaue Erläuterung wird hier verzichtet, die Abbildung soll die niedrige Abstraktionsebene und die Komplexität von EDIF darstellen.

EDIF dient zum Erzeugen von Schaltkreisen auf einem niedrigen Abstraktionsniveau. Ziel dieser Arbeit ist es jedoch, auf höheren Abstraktionsebenen zu arbeiten. VHDL bietet ein solches Niveau, weshalb diese Hochsprache zur Beschreibung von digitalen Schaltkreisen verwendet wurde und nicht das EDIF.

Kapitel 3

Verwendete Technologien

Im folgenden Kapitel werden die verwendeten Technologien für die Implementierung vorgestellt. Das auf Eclipse basierende Projekt Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), welches am Lehrstuhl Echtzeitsysteme und Eingebettete Systeme der Christian-Albrecht-Universität zu Kiel entwickelt wird, dient hierbei als Projektgrundlage. Das KIELER-Projekt stellt Funktionen und Transformationen bereit, die im Laufe der Arbeit verwendet wurden. Die Entwicklungsumgebungen Eclipse wurde verwendet, da die Entwicklung von KIELER auf Eclipse basiert. Die benötigten Transformationen von SCCharts zur Hardwarebeschreibung wurden in Xtend geschrieben und zum KIELER Projekt hinzugefügt. Als Hardwarebeschreibungssprache wurde die Very High Speed Integrated Circuit Description Language (VHDL) verwendet. Der von den Transformationen erzeugte VHDL Code wurde mit dem Xilinx[1] Tool Integrated Software Environment (ISE) nach Hardware übersetzt. Für Simulationszwecke und zum Sicherstellen der Richtigkeit der Transformationen, wurde das Simulationstool ISE Simulator (ISim), ebenfalls von der Firma Xilinx, verwendet. Die Tools der Firma Xilinx wurden verwendet, da am Lehrstuhl eine Evaluationsboard von Xilinx vorhanden war, auf dem Test durchgeführt werden konnten.

3.1 Eclipse

Eclipse[2] ist eine quelloffene Entwicklungsumgebung. Sie ist der Nachfolger von Visual Age for Java 4.0, dessen Quellcode im Jahr 2001 von IBM freigegeben wurde. Seit 2004 ist die Eclipse Foundation eine eigenständiges Unternehmen. Die wohl bekannteste Java integrierte Entwicklungsumgebung (IDE) Eclipse unterstützte ursprünglich nur die Entwicklung Java basierter Programme. Mittlerweile ist Eclipse jedoch vielseitig einsetzbar und unterstützt weitere Programmiersprachen wie C oder visuelle Sprachen wie die Unified Modeling Language (UML). Das Plugin Konzept von Eclipse trägt zur Vielseitigkeit bei. Eclipse selbst stellt nur eine Kernkomponente zur Verfügung, die die einzelnen Plugins lädt und verwaltet. Geladene Plugins erweitern die Funktionalität. Die Plugins und Eclipse sind in Java programmiert. Grafische Oberflächen werden mit SWT erstellt und für die Darstellung der GUI-Komponenten wird auf die nativen Komponenten des jeweiligen Betriebssystems zurückgegriffen.

[1] http://www.xilinx.com/
[2] http://www.eclipse.org/

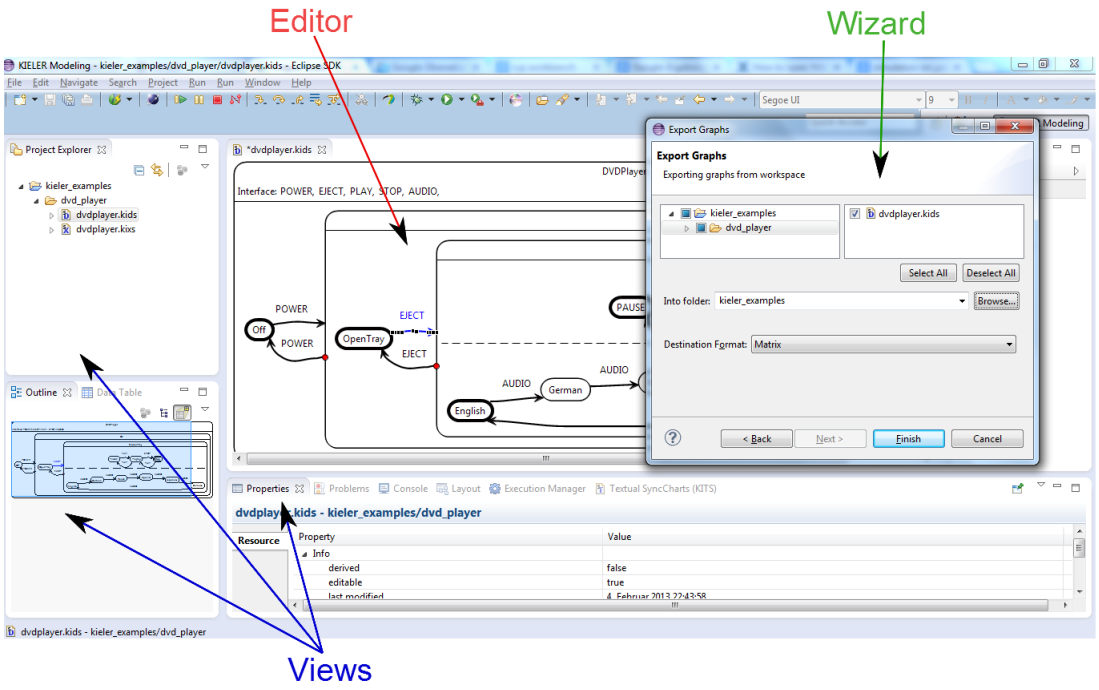


Abbildung 3.1. Eine Perspektive einer Eclipse Workbench [Har13]

Somit stellt Eclipse eine Rich Client Platform (RCP) zu Verfügung, die durch gezieltes Laden verschiedener Plugins auf die Lösung verschiedenster Probleme zugeschnitten werden kann. Durch eine wohldefinierte Schnittstellenbeschreibung ist es ebenfalls möglich, Plugins anderer Hersteller zu laden und zu verwenden. Eclipse besteht dabei im Wesentlichen aus *Sichten (Views)*, *Editoren* und *Perspektiven*, die in einer *Workbench* zusammengefasst werden. Eine Eclipse Workbench ist in Abbildung 3.1 zu sehen.

Bei *Views* handelt es sich um Fenster, die Informationen zum aktuellen Projekt liefern. Zu den *Views* zählen unter anderem der Projekt-Navigator, der Klassenexplorer und das Suchfenster. Die Editoren stellen die Möglichkeit bereit, Quellcode anzuzeigen und zu modifizieren. Eclipse stellt in der Regel für jede Programmiersprache einen eigenen textuellen oder grafischen Editor zu Verfügung. Die Anordnung und Auswahl angezeigter *Views* und *Editoren* kann frei gewählt werden. Diese Anordnung und Sichtbarkeit wird in Eclipse als *Perspektive* bezeichnet. Perspektiven können beliebig gespeichert und geladen werden. Der in Abbildung 3.1 gezeigte *Workbench* stellt gleichzeitig eine *Perspektive* dar. In der Abbildung ist ebenfalls ein *Wizard* dargestellt. Benutzer können in einen *Wizard* strukturiert Informationen eingeben um eine bestimmte Aufgabe auszuführen. Der gezeigt *Wizard* ist zum Exportieren von Graphen.

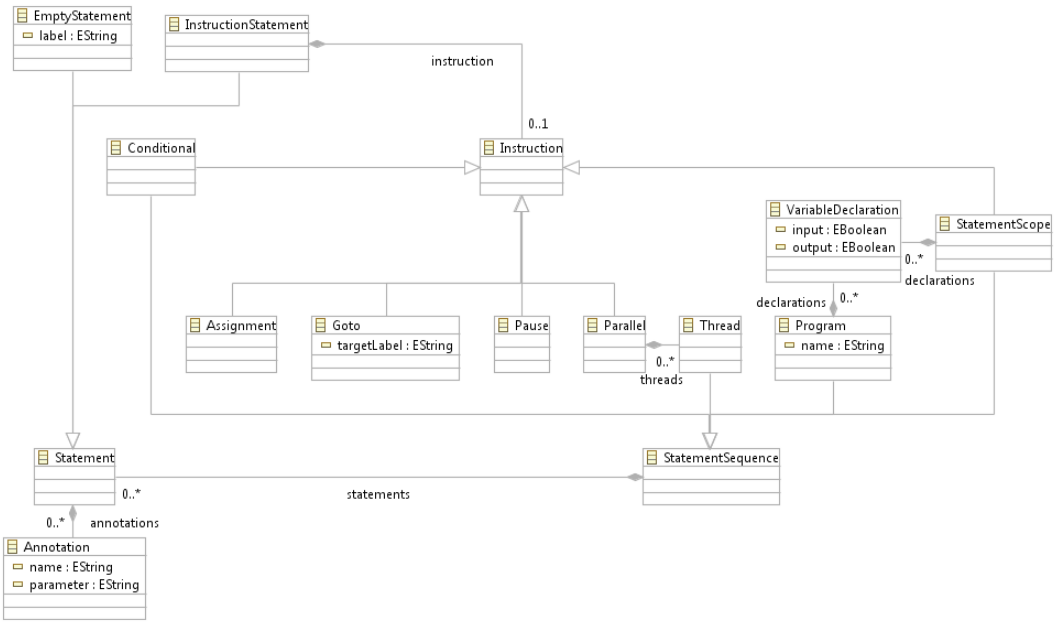


Abbildung 3.2. Beispiel eines Meta Modells — SCL Meta Modell

3.1.1 Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF)[3] ist ein Plugin für Eclipse. Es stellt in Eclipse die Möglichkeit bereit, modellgetrieben zu entwickeln. Das EMF wird verwendet, da SCCharts und die verwendeten Programmiersprachen in Modellen abgebildet sind.

Bei modellgetriebener Entwicklung handelt es sich um Techniken, die aus einem formal beschriebenen Model automatisch Software oder Hardware generiert. Das EMF generiert aus einem in XML Metadata Interchange (XMI) spezifiziertem Modell Java Code. Die erzeugten Klassen spiegeln das Modell in Java wider. Es wird ebenso Quellcode zum Anzeigen und zum Editieren das Modells generiert. Dabei besteht EMF im Wesentlichen aus drei Komponenten: *core EMF framework*, *EMF.Edit* und *EMF.Codegen*.

In EMF muss jedes Modell einem Meta-Modell entsprechen. Ein Meta-Modell ist eine abstrakte Syntax mit der Modelle beschrieben werden können. In Abbildung 3.2 ist ein Meta-Modell von SCL gezeigt. SCL ist eine textuelle Sprache zum Beschreiben von SCCharts. Die *core EMF framework* Komponente stellt unter anderem das Meta-Modell, eine effiziente API zum Manipulieren, sowie eine Serialisierung im XMI Format zum persistenten Speichern von Modellen bereit. Editoren für ein Modell können mit *EMF.Edit* erstellt werden. Ein Editor mit grafischer Oberfläche und vollem Funktionsumfang kann mit *EMF.Codegen* erzeugt werden.

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

```
3  class Templates {
4
5      def createVhdlComponent(String componentName, boolean setReset){
6          '''
7          component «componentName»
8              port( A: in std_logic;
9                    B: in std_logic;
10         «IF(setReset)»
11             «'''          reset: in std_logic;'''»
12         «ENDIF»
13                   S: out std_logic;
14                   Cout: out std_logic );
15         end component;
16         '''
17     }
18 }
```

(a) Xtend Template Code

```
1  COMPONENT HALBADDIERER
2     PORT( A: IN std_logic;
3           B: IN std_logic;
4           reset: IN std_logic;
5           S: OUT std_logic;
6           Cout: OUT std_logic );
7  END COMPONENT;
```

(b) Erzeugter VHDL Code

Abbildung 3.3. Beispiel für das Xtend Template

3.1.2 Xtend

Xtend[4] ist ein flexibler und ausdrucksstarker Dialekt von Java. Beim Compilieren wird Xtend-Quellcode in lesbaren Java-Quellcode übersetzt. Xtend ist dabei vollständig kompatibel zu Java.

Bei Xtend handelt es sich um eine streng getypte Programmiersprache die Java's Typsystem vollständig unterstützt. Ebenso können vorhandene Java Bibliotheken verwendet werden. Xtend erweitert Java und stellt hilfreiche Features, wie zum Beispiel Lambda-Ausdrücke, Operator-Überladung und Typen-Inferenz bereit.

Xtend kann in Eclipse integriert werden. Bei der Hardwaresynthese von SCCharts wird Xtend verwendet, um die benötigten Transformationen zu implementieren. Xtend eignet sich für Model zu Model (MtM) Transformation ebenso wie für die Erzeugung von textuellen Quellcode.

Die *Template Expressions* von Xtend stellen eine erweitertes String Konzept dar, die sogenannten *Rich Strings*. So bieten *Templates* die Möglichkeit Strings über mehrere Zeilen zu definieren, einen String in Abhängigkeit einer Bedingung oder mit Hilfe einer Schleife zu erstellen. Des Weiteren gibt es einen intelligenten Umgang mit Leerzeichen, welcher vorteilhaft für die Erzeugung textuell strukturierten, lesbaren Quellcodes ist.

Die *Templates* finden in dieser Arbeit Anwendung in der Generierung von VHDL-Quellcode. In Abbildung 3.3 werden beispielhaft die Funktion einiger *Templates* dargestellt. Die während der Programmierung eingefügten Leerzeichen werden auch im transformierten Quellcode abgebildet. Ein übersichtlicher und strukturierter Quellcode ist das Ergebnis.

[4] http://www.eclipse.org/xtend/documentation.html

3.2 Das JUnit Framework

JUnit[5] ist ein Framework zum Testen von Java-Programmen. Er ist besonders für automatisierte Tests einzelner Klassen oder Methoden geeignet. Mittlerweile gibt es ähnliche Konzepte für viele andere Programmiersprachen. Bei einem JUnit-Test können nur zwei Ergebnisse entstehen, der Test ist bestanden oder fehlgeschlagen. Das wird mittels spezieller *Exceptions* signalisiert.

JUnit wurde verwendet, um die Transformationen automatisch zu testen. Für jedes zu testende SCChart muss eine Testdatei angefertigt werden, die Signale der Umgebung emittiert und erwartete Ausgabesignale das Modells überprüft. Model und Testdatei werden in einem JUnit-Test gegeneinander validiert. Dieser Test wurde mit mehreren Modellen und Testdateien unternommen, um die Richtigkeit der Transformationen während der Entwicklung zu gewährleisten.

3.3 Kiel Integrated Environment for Layout Eclipse Rich Client

Das Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) Projekt soll die Handhabung und den Umgang beim Designen komplexer grafischer modellbasierter Entwürfe vereinfachen. So steht zum Beispiel die ständige Entwicklung automatischer Layouts diverser grafischer Komponenten im Fokus. Ebenso ist die Entwicklung neuer Methoden und Konzepte für das Editieren und das Arbeiten an Modellen und die dynamische Visualisierung von Diagrammen ein Forschungsschwerpunkt. Die dynamische Visualisierung wird zum Beispiel für die Simulation von Diagrammen verwendet.

KIELER wird mit Hilfe von Eclipse entwickelt und basiert auf der Rich Client Platform. Dabei integriert KIELER einen Großteil an Modellierungsprojekten von Eclipse (EMF,GMF,Xtext, etc.). Bei KIELER handelt es sich um ein *Open-Source-Projekt*, welches unter der *Eclipse Public Licencse* entwickelt wird. Das KIELER-Projekt gliedert sich, wie in Abbildung 3.4 zu sehen, in die vier Teilgebiete *Semantics*, *Pragmatics*, *Layout* und *Demonstrators (Demos)* auf.

Semantik

Das Teilgebiet der *Semantik* stellt eine Infrastruktur bereit, in der Ausführungssemantiken für Meta-Modelle definiert werden. Dabei kann es sich zum Beispiel um einen Simulator, basierend auf C oder Ptolemy[6], handeln. Weitere textuelle oder grafische Simulatoren können über den KIELER Execution Manager (KIEM)[7] eingebun-

[5] http://www.junit.org/
[6] http://ptolemy.eecs.berkeley.edu/
[7] http://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=328095

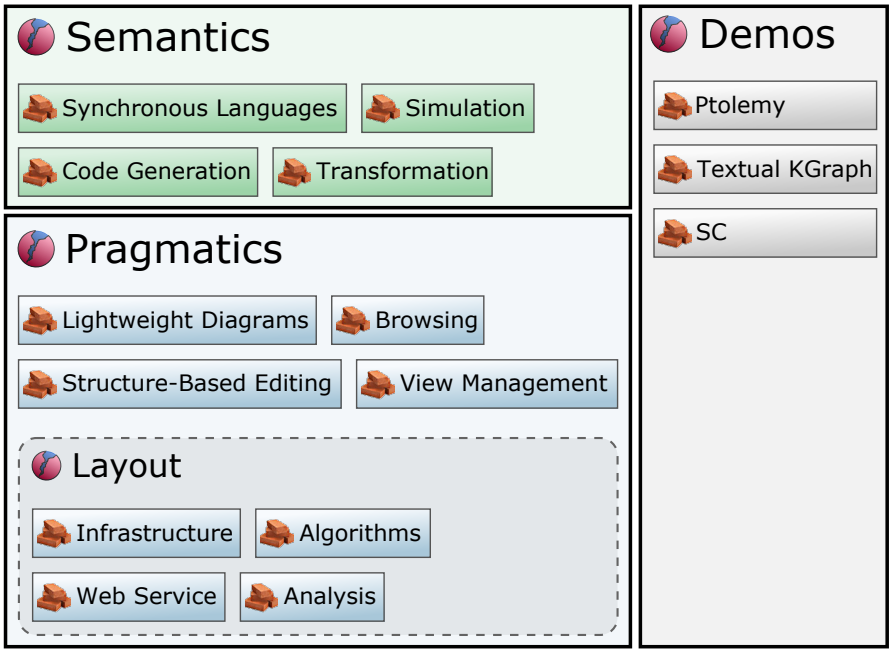


Abbildung 3.4. Übersicht des KIELER Projektes [RK13]

den werden. Die Transformation von einer Quellsprache in eine andere Zielsprache gehört in den Bereich der Semantik. Die Transformationen von SCCharts zur Sequentially Constructive Language (SCL), welche in dieser Arbeit benötigt werden, gehören ebenfalls zur Semantik. Da es sich in dieser Arbeit ebenfalls um eine Transformation zwischen Sprachen handelt, ist diese Arbeit in das Teilgebiet der *Semantik* einzuordnen.

Pragmatik

Die *Pragmatik* beschäftigt sich mit den praktischen Aspekten des Programmierens im Hinblick auf modellgetriebene Entwicklung (Model-Driven Engineering (MDE)) [Sch06]. Das beinhaltet das Erstellen und Modifizieren von Modellen sowie die Synthese verschiedener Sichten auf Modelle. Das ist zum Beispiel für Gruppen mit unterschiedlichen Interessen an einem Modell von Bedeutung. Strukturbasiertes Editieren von Modellen, die auf einer abstrakten Syntax beruhen, stellt zum Beispiel das Plugin KIELER Structure-based Editing (KSbasE) zur Verfügung. Damit ist es möglich, neue Elemente zu erstellen, sie zu bearbeiten, zu verbinden oder zu löschen.

KIELER Lightweight Diagrams (KLighD) stellt die Möglichkeit einer Synthese von Modellelementen zu einer grafischen Repräsentation zur Verfügung. Ziel von KLighD ist es, eine transiente grafische Repräsentation eines Modells zu generieren, welche den individuellen Ansprüchen des Modellierers entspricht.

Die *View-Management*-Komponente stellt eine dynamische Visualisierung für den Benutzer bereit. So erfährt ein Element eines Modells, welches ausgewählt wird, zum Beispiel eine optische Hervorhebung. Es werden diverse *Trigger* (z.B. Auswahl eines Elementes) in Effekte umgesetzt (z.B. Hervorhebung des Objektes).

Layout

Die Forschungen im Bereich *Layout* sollen dem Benutzer das Erstellen und Editieren von Layouts erleichtern. Teilweise müssen verschiedene Änderungen an einem Layout gemacht werden, bevor ein neues Element einfügt werden kann. Zum Beispiel muss Raum für ein neues Element geschaffen werden. Das Weiteren müssen für ein übersichtliches und gut lesbares Layout einzelne Komponenten (Knoten und Kanten) manuell angeordnet werden. Diese Schritte benötigen Zeit und hindern den Entwickler an einem effektiven, schnellen Design.

Im Bereich *Layout* wird an Optimierungen für den Modellierer geforscht. So übernehmen automatische Layouts und Algorithmen diese Aufgaben. KIELER Infrastructure for Meta Layout (KIML) ist die Kernkomponente, sie stellt die Schnittstelle zwischen dem grafischen Editor und dem jeweiligen Layoutalgorithmus dar. Es werden sowohl Open-Source Layoutalgorithmen, als auch spezialisierte Algorithmen (KIELER Layouters (KLay)) bereitgestellt.

Demonstrators

Die *Demonstrators* beinhalten diverse Editoren. Die Editoren werden verwendet, um die entwickelten Technologien der anderen Teilbereiche zugänglich zu machen und zu validieren. Zu den Editoren zählen unter anderen der *Yakindu*-Editor [Har13], welcher den *ThinkCharts*-Editor ersetzt, zum Editieren von SyncCharts und der *KGraph*-Editor, zum textuellen Beschreiben von *KGraph*-Graphen.

3.4 Die Hardwarebeschreibungssprache VHDL

Eine Hardwarebeschreibungssprache ermöglicht es, digitale Systeme textuell zu beschreiben. Zwei bekannte Hardwarebeschreibungssprachen sind VHDL und Verilog. Verilog ist vergleichbar mit einer Programmiersprache wie C, und VHDL ist an *Ada* angelehnt. Die IEEE hat 1987 VHDL als einen IEEE-Standard festgelegt und Verilog im Jahr 1995.

VHDL entstand durch die US-amerikanische Regierung die 1980 die *VHSIC*-Initiative ins Leben gerufen hat. Sie sollte den technologischen Rückstand zwischen der Regierung und der Privatwirtschaft aufholen. Das US-Verteidigungsministerium forderte, dass alle entwickelten Schaltungen nach dem 30. September 1988 in VHDL dokumentiert sein mussten, damit war VHDL als Standard etabliert. Es wurde zum

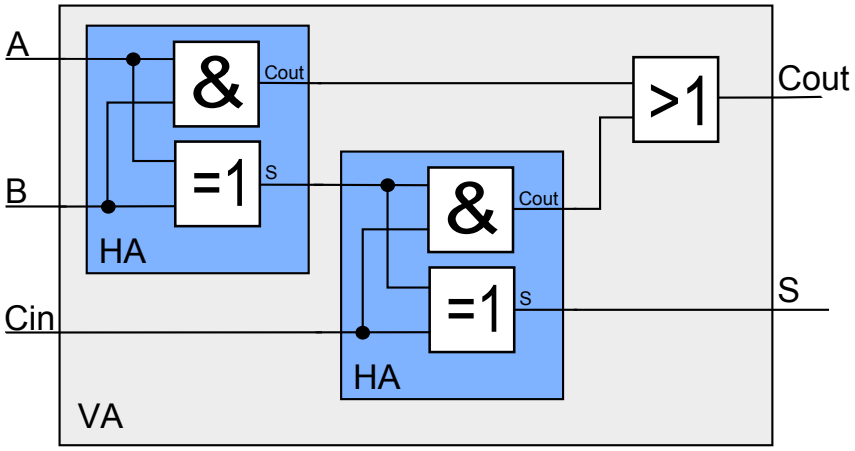


Abbildung 3.5. Komponenten Sicht eines 2 Bit Addierers

ersten Mal 1987 durch die IEEE standardisiert. Die aktuelle Standardisierung, die weitere Sprachergänzungen enthält, ist von 1993.

3.4.1 Die Programmierung mit VHDL

VHDL dient zur textuellen Beschreibung von digitalen Systemen. Mit VHDL ist es möglich, Schaltkreise auf einer abstrakten Ebene zu beschreiben. So gibt es eine externe und eine interne Sicht auf die beschriebenen VHDL-Komponenten [Bha92]. Bei der externen Sicht handelt es sich um die Sicht, die die Schnittstellenbeschreibung widerspiegelt, über welche die Komponente mit anderen Komponenten kommunizieren kann. Die interne Sicht beschreibt das Verhalten bzw. die Struktur. Das Beispiel in Abbildung 3.5 soll die verschiedenen Sichten verdeutlichen.

Es handelt sich in dem Beispiel um einen Voll-Addierer (VA), der aus zwei Halb-Addierern (HA) besteht. Die interne Sicht eines Halb-Addierers beschreibt sein Verhalten, während die interne Sicht des Voll-Addierers die Struktur und das Verhalten des gesamten Addierers als Komponente beschreibt. Die externe Sicht des Voll-Addieres beschreibt hingegen die Schnittstellen, die die Komponente Voll-Addierer besitzt. Zur Schnittstellenbeschreibung gehören alle Ein- und Ausgabesignale (A, B, Cin, Cout und S) des Addierers. Mit VHDL ist es möglich, einzelne Komponenten untereinander zu verbinden, Komponenten mehrfach zu verwenden (siehe Halb-Addierer) und hierarchische Modelle zu entwickeln. Es werden durch VHDL somit Möglichkeiten zum verhaltens- und strukturbasierten Programmieren gegeben.

3.4.2 Simulation von Hardwarebeschreibungen

Um das erwartete Verhalten eines synthetisierten Hardware-Modells zu validieren, können Modelle simuliert werden. Dazu wird eine *Testbench* benötigt. Diese simuliert

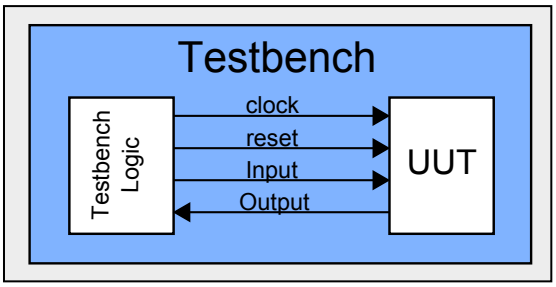


Abbildung 3.6. Abstrakte Sicht auf eine VHDL Testbench, mit Testbench Logik und dem *Unit Under Test*

die Eingangssignale und überprüft das beschriebene Verhalten der Ausgangssignale. VHDL erlaubt es, die Testbench auch in dieser Sprache zu schreiben.

Eine Testbench selbst besitzt keine Ein- oder Ausgänge. Sie bietet nur die Möglichkeit der Simulation und ist nicht synthetisierbar [Inc12b]. In der Testbench wird das zu testende Model instantiiert, die sogenannte Unit Under Test (UUT). Die UUT ist ein in VHDL beschriebenes Hardwaremodul. Es können entsprechende Eingangssignale generiert werden, um das Modul zu testen. Die vom Modul erzeugten Ausgangssignale können gelesen und überprüft werden. Die Abbildung 3.6 zeigt einen schematischen Aufbau einer Testbench inklusive dem UUT.

Die Komponente *Testbech Logic* stammt aus der Beschreibung der Testbench und simuliert, entsprechend der Beschreibung, die Signale clock, reset und Input. Das UUT reagiert entsprechend seiner Beschreibung und erzeugt die Output-Signale, welche von der *Testbench Logic* auf erwartete Signale hin überprüft werden kann.

3.5 ISE Design Umgebung

Das Integrated Software Environment (ISE) ist eine Entwicklungsumgebung zur Synthese und Analyse von VHDL Designs. Entwickelt wird es von der Firma Xilinx. Es ermöglicht dem Programmierer seine Hardwarebeschreibung zu synthetisieren (kompilieren), Register Transfer Logic (RTL)-Diagramme zu untersuchen, *Timing*-Analysen durchzuführen und die beschriebene Komponente auf einer ausgewählten Zielhardware zu synthetisieren.

Xilinx stellt ISE in drei Varianten zu Verfügung, *ISE Design Suite: Embedded Edition*, *ISE Design Suite: System Edition* und *ISE Design Suite: WebPACK Edition*. Verwendet wurde die *WebPACK Edition*, sie ist eine frei verfügbare, jedoch eingeschränkte Version. Es stehen nur ausgewählte Xilinx FPGAs zur Synthese zur

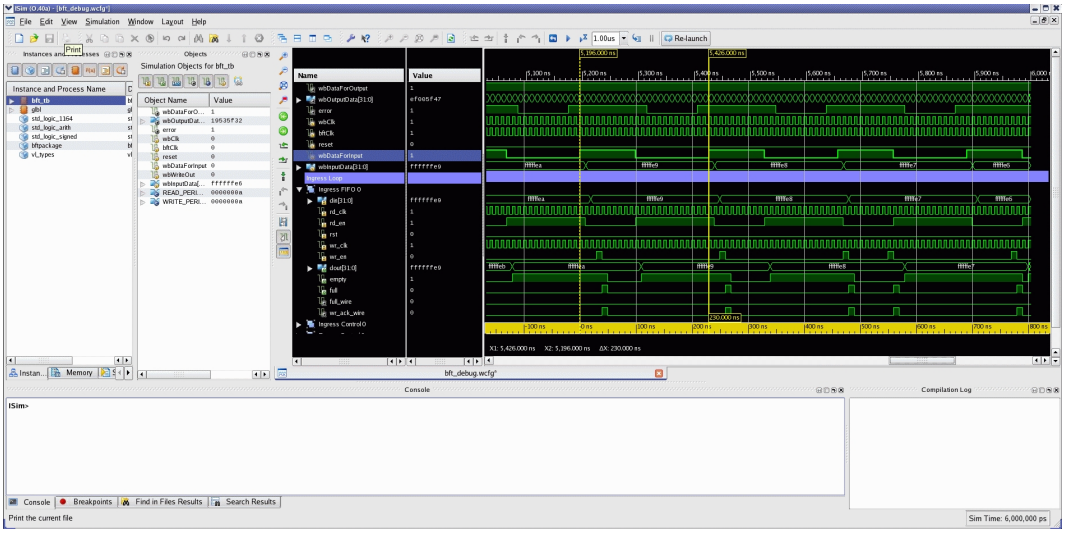


Abbildung 3.7. ISE-Simulator mit einem Signaldiagramm

Verfügung. Sie bietet jedoch den vollen Funktionsumfang. Um das *Evaluationboard* aus Abschnitt 6.2.3 zu programmieren, wird eine vollständig lizenzierte Version benötigt, da dieser FPGA nicht von der freien *WebPACK Edition* unterstützt wird.

ISE wurde verwendet, um den transformierten VHDL-Code zu synthetisieren und zu verifizieren. Des Weiteren wurde die Entwicklungsumgebung verwendet, um transformierte SCCharts auf einem FGPA zu synthetisieren, somit konnte das Ergebnis der Transformation auf “echter“ Hardware überprüft werden.

3.5.1 ISE Simulator

Beim ISE Simulator (ISim) handelt es sich um einen umfangreichen und voll funktionsfähigen HDL-Simulator[8]. ISim bietet unter anderem die Möglichkeit, Signalverläufe der simulierten Hardwarebeschreibung anzuzeigen.

Der Simulator kann mit der *Debug*-Umgebung von Eclipse verglichen werden. Es können zeitliche Verläufe verschiedener Signale überprüft werden und das Verhalten kann analysiert werden. Die Abbildung[9] 3.7 zeigt die Oberfläche der Simulationsumgebung. Im linken Bereich können Signale ausgewählt werden, die im rechten Bereich angezeigt werden. Signale können hervorgehoben, ausgeblendet, verglichen und in Gruppen zusammengefasst werden.

Der ISE Simulator wurde verwendet, um während der Entwicklung der Transformationen das Verhalten der Schaltkreise zu simulieren und zu überprüfen. Die *Timing Analyse* von ISim wurde zur Analyse von Fehlern verwendet und um die korrekte Funktionsweise des Schaltkreises zu überprüfen.

[8] http://www.xilinx.com/tools/isim.htm

[9] Abbildung von: http://www.xilinx.com/tools/isim.htm

Kapitel 4

Transformation

In diesem Kapitel werden die Ideen und Konzepte der einzelnen Transformationen der Hardwaresynthese aus SCCharts vorgestellt. Zu Beginn werden die verwendeten Sprachen in Unterkapitel 4.1 erläutert. Danach ist das Kapitel in die einzelnen Transformationsschritte gegliedert.

Für die Transformation von *SCCharts* zu Hardware werden *Core SCCharts* als Ausgangsmodell erwartet. SCCharts können auch in einer erweiterten Variante, den *extended SCCharts* vorliegen. Diese müssten in einem vorbereitenden Schritt in eine Core Variante transformiert werden, siehe Abschnitt 4.2.1. Diese Transformation ist jedoch nicht Teil dieser Arbeit.

Die vorliegenden Core SCCharts werden anschließend in eine SCL Repräsentation überführt (Abschnitt 4.2.2), diese wiederum kann als *Sequentially Constructive Graph (SCG)* visualisiert werden. Auf Basis des SCG werden Analysen durchgeführt. Diese Analysen bestimmen, ob es für ein Programm ein gültiges *Schedule* gibt. Sofern das Programm ausführbar ist, wird es in sequentielles SCL transformiert. Sequentielles SCL besitzen keine parallelen bzw. nebenläufigen Regionen mehr. Die beschriebene Transformation ist Teil der Diplomarbeit von Herrn Smyth [Smy13] und wird in Abschnitt 4.2.5 vorgestellt.

Basierend auf dem sequentiellen SCL Modell können nun zwei Transformationsansätze zu VHDL verfolgt werden. Die erste naive Transformation (Unterkapitel 4.3) übersetzt SCL-Programme nahezu direkt in VHDL. In einer zweiten Transformation (Unterkapitel 4.4) wird auf dem SCL-Programm das Static Single Assignment (SSA) angewandt, eine Möglichkeit Datenabhängigkeiten aufzulösen, bevor es nach VHDL übersetzt wird. Die VHDL Synthese wird mit Hilfe des Xilinx Werkzeugs ISE durchgeführt.

In der Übersicht in Abbildung 4.1 sind die beschriebenen Wege von SCCharts zur Hardwaresynthese illustriert. Es ist weiterhin möglich, sequentiellen SCL-Code zu C oder Java zu übersetzen, um die Ausführung auf herkömmlichen Prozessoren zu verwirklichen.

4.1 Die verwendeten Sprachkonzepte

Die Transformation von SCCharts zu Hardware wird in mehreren Schritten durchgeführt. In den einzelnen Phasen werden verschiedene Sprachkonzepte verwendet.

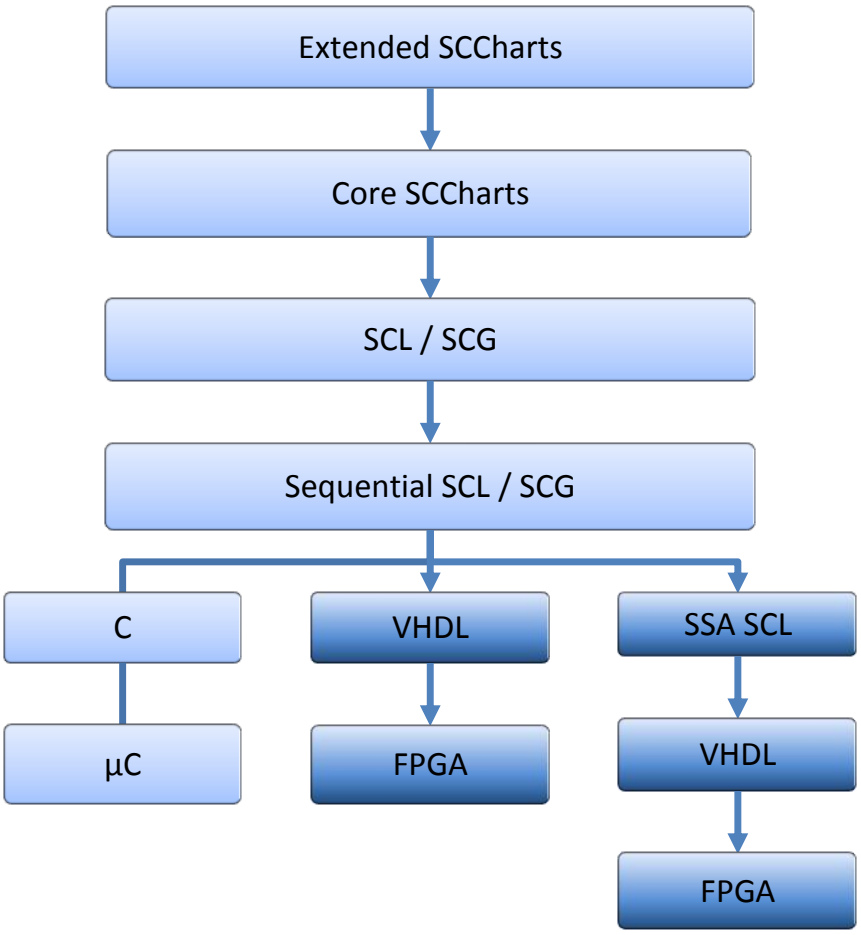


Abbildung 4.1. Transformationsschritte von extended SCCharts zu Hard- und Software

Damit die verschiedenen Transformationen nachvollzogen werden können, ist eine nähere Betrachtung der einzelnen Sprachkonzepte nötig. Im Folgenden werden die benötigten Sprachkonzepte und das verwendete Ausführungsmodell näher erläutert.

4.1.1 Sequentielle Konstruktivität

Die Esterel/Berry Semantik besitzt einige Einschränkungen zum Erhalt des Determinismus, welche dem Entwickler nicht intuitiv erscheinen und nicht zwangsläufig nötig sind. In den Programmiersprachen wie C oder Java ist das Konstrukt `if ! i then i = true end;` eine gültige Anweisung. Dieses Konstrukt ist unter der Berry Semantik nicht erlaubt, obwohl es ohne Probleme ausführbar wäre. Diese Anweisung ist ungültig, da sie nicht dem Signalkohärenz Gesetz, siehe Abschnitt 1.1.1, entspricht, welche in der Berry Semantik verwendet wird. Es ist nicht erlaubt, dass ein Signal zwei Werte

innerhalb eines Ticks hat. Mit einer Auflösung dieser Einschränkung haben sich von Hanxleden et al. im Artikel “Sequentially Constructive Concurrency - A Conservative Extension of the Synchronous Model of Computation” [vHMA+13b] auseinandergesetzt und präsentieren das *Sequentially Constructive Model of Computation (SC MoC)*.

Das Ziel ist es, die Einschränkung der Konstruktivität der Berry Semantik im SC Ausführungsmodell aufzuheben und weiterhin ein deterministisches Verhalten zu garantieren. Im Gegensatz zur Berry Semantik muss das SC MoC nebenläufige Mehrfachzuweisungen zu Variablen im selben Tick erlauben, solange das Modell ohne Konflikte beim nebenläufigen Variablenzugriff ausgeführt werden kann.

Nebenläufiger Variablenzugriff

Um eine konfliktfreie Ausführung nebenläufiger Variablenzugriffe zu garantieren, müssen nach von Hanxleden et al. [vHMA+13c] die Variablenzugriffe kategorisiert und geordnet werden. So müssen Schreibzugriffe auf Variablen ausgeführt werden, bevor Variablen gelesen werden. Schreibende Zugriffe können wie folgt unterschieden werden:

Konfluentes Schreiben: Zwei nebenläufige Zuweisungen zur gleichen Variable sind *konfluent*, wenn die Reihenfolge, in der die Zuweisungen ausgeführt werden, unerheblich ist. Die Variable hat, egal in welcher Reihenfolge die Zuweisungen ausgeführt wurden, den gleichen Wert.

Relatives Schreiben: Relative Schreibzugriffe haben die Form $x = f(x, e)$, wobei x eine Variable und e ein Ausdruck ist, welcher x nicht referenziert, zum Beispiel $i = i||true$. Die Funktion f muss die folgende Eigenschaft erfüllen: Für eine Funktion (*combination function*) $f(x, y)$ auf x gilt, für alle x und alle y_1,y_2, $f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$.

Absolutes Schreiben: alle Schreibzugriffe, die nicht relative Zugriffe sind, sind absolute Schreibvorgänge, zum Beispiel $i = true$.

Die Ordnung nicht konfluenter nebenläufiger Variablenzugriffe unterliegt dem sogenannten “initialise-update-read”-Protokoll. Es besagt, dass in jedem Tick, bei allen nebenläufigen Zugriffen auf Variablen, zuerst absolute Schreibzugriffe vor relativen Schreibzugriffen ausgeführt werden müssen. Nebenläufiges absolutes Schreiben auf dieselbe Variablen muss dabei konfluent sein. Ansonsten kann kein gültiges Schedule gefunden werden und das Programm muss abgelehnt werden. Zuletzt können Variablen gelesen werden.

Nach von Hanxleden et al. [vHMA+13c] können zwei Definitionen getroffen werden, die ein Modell erfüllen muss, um unter dem SC MoC ausgeführt werden zu können.

Definition 1. Eine SC-zulässige Ausführung ist eine Ausführung des Programms, welche auf dem “initilise-update-read”-Protokoll für nicht konfluente Variablenzugriffe beruht.

Definition 2. Ein Programm ist sequentiell konstruktiv (SC) wenn (i) eine SC-zulässige Ausführung existiert und (ii) jede zulässige Ausführung für jeden Tick dasselbe Ergebnis generiert (Determinismus).

4.1.2 SCCharts

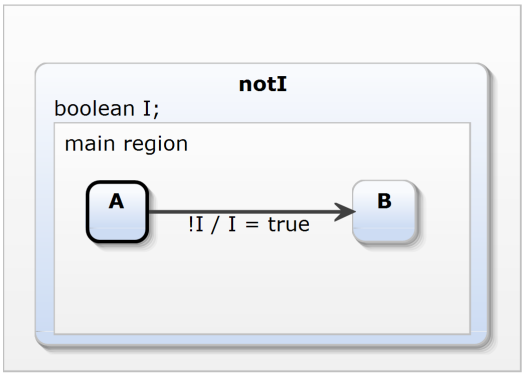
Bei *Sequentially Constructive Statecharts*, kurz *SCCharts*, handelt es sich um eine Erweiterung von SyncCharts [vHMA+13a]. SCCharts sind eine visuelle Sprache, die nicht wie SyncCharts auf der Konstruktivität von Berry basieren, sondern auf dem sequentiell konstruktiven Ausführungsmodel (SC MoC) basiert, siehe Abschnitt 4.1.1, und sind speziell dafür entwickelt worden, sicherheitskritische reaktive Systeme zu modellieren. Wie SyncCharts weisen auch SCCharts ein deterministisches nebenläufiges Verhalten auf. Das sequentiell konstruktive Ausführungsmodell regelt den nebenläufigen Zugriff auf geteilte Variablen und garantiert Determinismus im SCChart. SCCharts verwenden im Gegensatz zu SyncCharts getypte Variablen anstelle von Signalen. In einer erweiterten SCChart Variante können Signale immer noch verwendet werden. Weiterhin erlaubt das SC-MoC für SCCharts gegenüber SyncCharts mehrfache Zuweisungen zu Variablen im selben Tick, solange für das Programm ein gültiges Schedule existiert. Dafür wurde das limitierte Signalkohärenz Gesetz, Abschnitt 1.1.1, angepasst, um mehrere Werte von Variablen in einem Tick zuzulassen. So ist zum Beispiel die Anweisung in Abbildung 4.2 im synchronen Ausführungsmodell für SyncCharts nicht zugelassen, wenn I den Wert false führt und I im selben Tick ebenfalls der Wert true zugewiesen wird. Diese Anweisung ist im sequentiell konstruktiven Ausführungsmodel allerdings zugelassen, da sich das Programm dort problemlos nach der Definition ausführen lässt.

Das Verhalten von SCCharts kann mit einer minimalen Anzahl von ausdrucksstarken Elementen modelliert werden, diese bilden die sogenannten *Core SCCharts*. Core SCCharts bestehen dabei im Wesentlichen aus Zustandsautomaten und nebenläufigen Programmierkonstrukten. *Extended SCCharts* sind eine Form der SCCharts mit einer erweiterten Anzahl an komplexeren Modellierungskomponenten, sie enthalten zum Beispiel *Signale*. Sie erlauben dem Entwickler, komplexe Sachverhalte übersichtlicher und komprimierter zu modellieren.

Core SCCharts

Bei den Core SCCharts handelt es sich um die minimale Anzahl an ausdrucksstarken Elementen, die zum Ausdrücken vollständiger Modelle ausreichend ist. Im Folgenden

```
1 if ! I then I = true end
```



(a) Variable mit möglichweise zwei verschiedenen Werten

(b) SCChart mit einer Variablen die eventuell zwei Werte in einem Tick besitzt

Abbildung 4.2. Eine Variable die eventuell zwei Werte in einem Tick besitzt

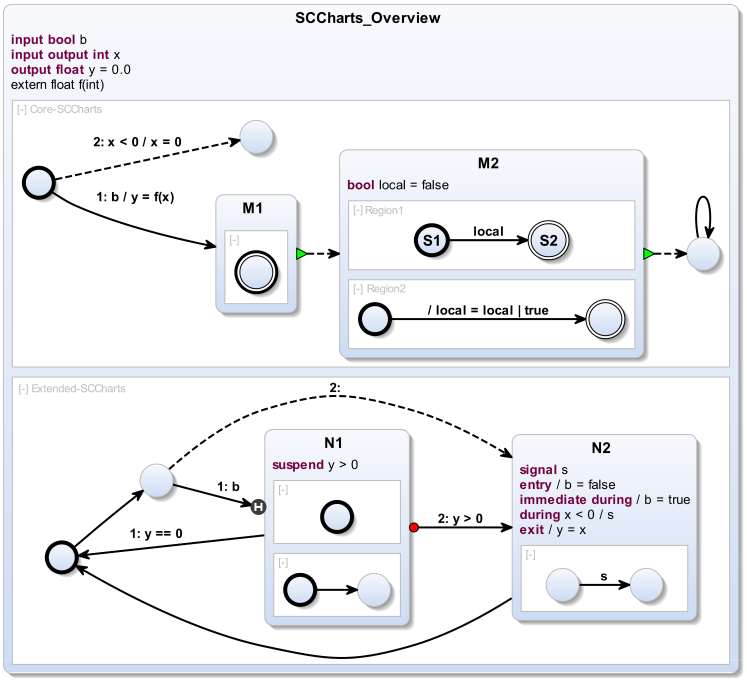


Abbildung 4.3. SCCharts Übersicht [MSvHM13]

wird ein Einblick in die Eigenschaften und Elemente von Core SCCharts gegeben. Abbildung 4.3 zeigt ein SCChart mit einer Reihe von Core und extended SCCharts Elementen, die nun näher erläutert werden. Im oberen Teil des SCCharts befinden sich die Core Elemente.

Zustände: SCCharts bestehen aus Zuständen (*states*) und Transitionen (*transitions*). Zustände können als initiale und finale Zustände markiert sein. Ein Zustand, der durch ein weiteres SCCharts verfeinert wird, welches aus einer oder mehreren Regionen bestehen kann, wird *makro state* (Makrozustand) genannt. Alle anderen Zustände sind einfache (*simple*) states. Ein Zustand ist *aktiv*, wenn der Kontrollfluss des SCCharts sich in ihm befindet. Ein Zustand ist transient, wenn er betreten und im selben Tick wieder verlassen wird.

Transitionen: Transitionen gehen von einem Anfangs-Zustand zu einem Ziel-Zustand. Eine Transition kann aus einem *Trigger*, der einer Bedingung entspricht, bei deren Erfüllung die Transition genommen wird und einem *Effect*, der eine Aktion im Modell hervorruft, bestehen. Beide sind optional. Transitionen von einem simple state, werden genommen und der Effect ausgeführt, wenn der Trigger zu true ausgewertet wird. Transitionen von makro states werden genommen, wenn alle internen Regionen in den finalen Zuständen sind, sie heißen *normal termination*. Grundsätzlich sind Transitionen verzögert, d.h. Transitionen können frühestens im nächsten Tick, nachdem der Zustand betreten wurde, ausgeführt werden. Eine gestrichelte Linie symbolisiert eine instantane Transition. Sie wird im selben Tick, indem der Zustand betreten wurde, ausgewertet.

Interfacebeschreibung: Die Interfacebeschreibung befindet sich am oberen Ende eines SCCharts, in ihr werden Variablen und externe Funktionen deklariert. Variablen können Eingabevariablen, welche Eingaben an das SCChart erlauben, oder Ausgabevariablen, welche Werte an die Umgebung liefern, sein. Auf der äußersten Ebene eines SCCharts bedeutet das, dass Eingabevariablen von der Umgebung zu Beginn eines Ticks gelesen und Ausgaben am Ende eines Ticks vom SCCharts geschrieben werden, siehe Abbildung 1.2. Weiterhin können in der Interface Deklaration auch lokale Variablen, welche weder Ein- noch Ausgang sind, erstellt werden.

Regionen: Regionen werden durch eine gestrichelte Linie angezeigt und sind nebenläufige SCCharts. Jede Region hat einen eindeutigen Start-Zustand. Wird ein End-Zustand einer Region betreten, terminiert diese.

Abbildung 4.4 zeigt ein Beispiel für Core SCCharts. Es ist das “Hello World”der sequentiell konstruktiven Welt zu sehen. Zu Beginn werden die Ausgabevariablen O1 und O2 mit false initialisiert. Im Makrozustand WaitAB wird auf die Eingaben A und B gewartet. Wird ein A im ersten Tick und ein B im zweiten Tick gesetzt, terminiert der Makrozustand. Ebenso, wenn im zweiten Tick nur A gesetzt wird. Wenn der Zustand WaitAB terminiert, wird O1 auf false und O2 auf true gesetzt und der Zustand ABO terminiert.

In dieser Arbeit werden für die Transformationen Core SCCharts vorausgesetzt. Dennoch soll ein kurzer Überblick über die extended SCCharts gegeben werden.

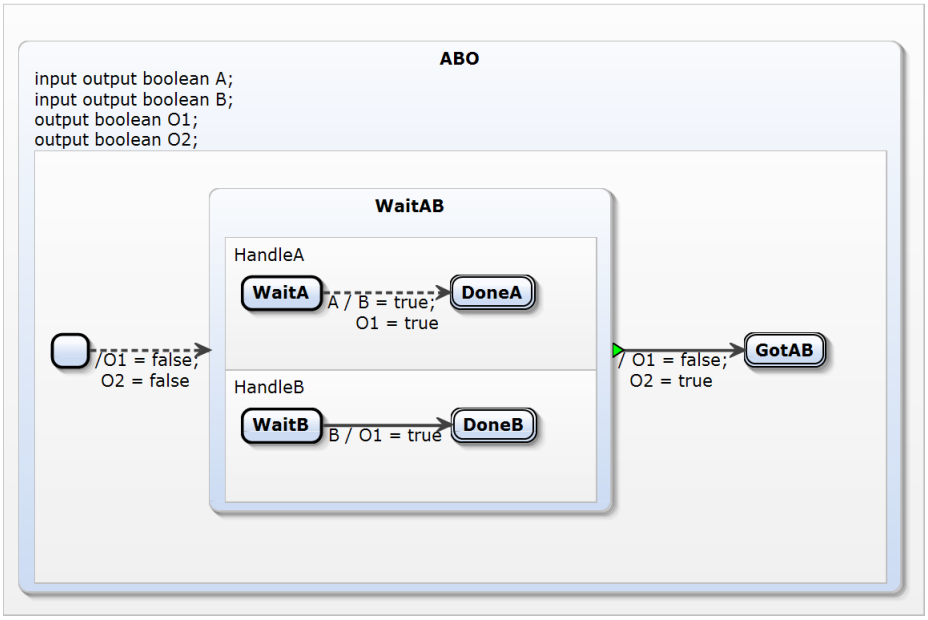


Abbildung 4.4. Das ABO Beispiel als SCChart modelliert

Werden extended SCCharts verwendet, können sie mit Hilfe von M2M Transformation in die Core SCChart Variante überführt werden.

Extended SCCharts

Bei extended SCCharts handelt es sich um eine erweiterte Version von Core SCCharts. Die Core SCCharts sind um zusätzliche syntaktisch angereicherte Elemente erweitert, mit denen komplexere Modelle einfacher umgesetzt werden können. Diese zusätzlichen Elemente können mit M2M Transformationen überschaubarer Komplexität [vHMA+13a] auf Core Elemente zurückgeführt werden. Im unteren Teil der Abbildung 4.3 ist ein SCChart mit extended Elementen dargestellt.

Connector: Konnektoren sind transiente Zustände und müssen in jedem Tick eine gültige Transition besitzen, die genommen werden kann. Sie werden oft als bedingte Entscheidungen (*Conditionals*) verwendet.

Strong Abort: *Strong Aborts* unterbrechen die Ausführung eines Makrozustandes (*Preemption*), wenn der Trigger genommen wird.

History Transition: *History Transitionen* kehren in den Zustand zurück, in dem der Makrozustand unterbrochen wurde, zum Beispiel durch ein strong abort.

Suspend: Ein *Suspend* stoppt die Ausführung des aktuellen Zustandes, solange die Bedingung gültig ist.

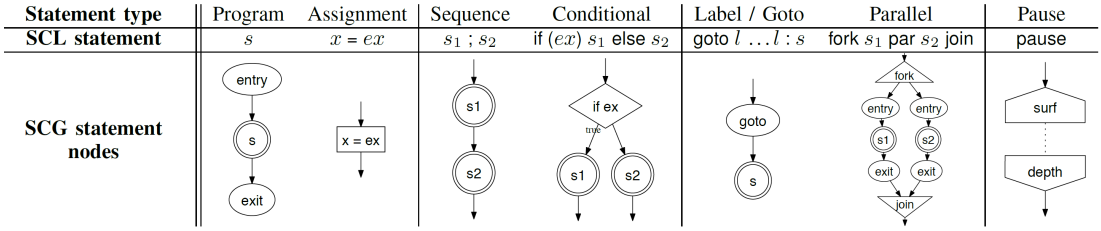


Abbildung 4.5. Übersicht der SCL und SCG Befehle [vHMA+13b]

Entry, During und Exit actions: Eine *Entry action* wird sofort beim Betreten einen Zustandes ausgeführt, eine *During action* hingegen in jedem Tick, in dem der Zustand aktiv ist und eine *exit action* wird beim Verlassen des Zustandes ausgeführt.

Pre: Der *Pre*-Operator gibt den Wert einer Variablen aus, dem vorhergehenden Tick zurück.

Signal: In Core SCCharts gibt es nur Variablen, das *signal* erweitert die SCCharts um das Signal Konzept aus Esterel [Ber00] und SyncCharts [And96]. Ein Signal kann während eines Ticks absent oder present sein, siehe Abschnitt 1.1.1.

4.1.3 Sequentially Constructive Language

Die *SC Language*, kurz *SCL*, ist eine nebenläufige, imperative, minimale Sprache, welche die Kommunikation über geteilte Variablen erlaubt. Um das sequentiell konstruktive Ausführungsmodell (SC MoC) darstellen zu können, haben von Hanxleden et al. die Sprache SCL [vHMA+13b] entwickelt. Die Syntax der Sprache basiert auf C, Java und Esterel. Da SCL und SCCharts auf dem selben Ausführungsmodell beruhen, können SCCharts mittels SCL textuell ausgedrückt werden. SCL-Konstrukte haben die folgende abstrakte Syntax für Anweisungen

$$s \quad ::= \quad x = e \mid s\ ;\ s \mid \text{if } (ex)\ s \text{ else } s \mid l : s \mid \text{goto } l \mid \text{fork } s \text{ par } s \text{ join} \mid \text{pause}$$

wobei x eine *Variable*, e ein *Ausdruck* und $l \in L$ ein *Programmlabel* ist [vHMA+13c].

Die Befehle, die SCL bereitstellt, sind in Abbildung 4.5 dargestellt. Des Weiteren sind grafische Repräsentationen der SCL-Befehle aufgeführt. Mittels der graphischen Elemente kann ein Kontrollflussgraph erstellt werden, der sogenannte Sequentially Constructive Graph (SCG). Der SCG ist ein Variante von SCL.

Assignments: Eine Zuweisung (*assignment*) der Form $x = ex$ schreibt einen Wert in die Variable x. In dem Ausdruck (*expression*) ex können weitere Variablenzugriffe vorhanden sein.

Sequence: Eine Sequenz (*sequence*) teilt zwei aufeinanderfolgende Anweisungen auf. Als Trennzeichen wird ein Semikolon verwendet.

Condtitional: Ein bedingter Sprung (*conditional*) wertet den Ausdruck ex aus und führt entsprechend die Anweisungen des Ausdrucks im *then* oder *else*-Zweig aus.

Labels: Labels markieren spezielle Programmpositionen und können frei platziert werden.

Goto: Für Sprünge wird die Anweisung goto verwendet. Eine goto-Anweisung springt ein definiertes Label an.

Parallel: Nebenläufigkeiten werden mit dem Konstrukt fork par join programmiert. Ein fork spaltet den Kontrollfluss in zwei separate Threads auf und ein join synchronisiert den Kontrollfluss, wenn beide Threads terminiert sind. Die Threads werden mit dem par-Ausdruck textuell voneinander getrennt.

Pause: Eine pause-Anweisung stoppt den Kontrollfluss des aktiven Threads für den aktuellen Tick.

Ein SCL-Programm ist *wohlgeformt*, wenn es die folgenden Kriterien [vHMA+13c] aufweist:

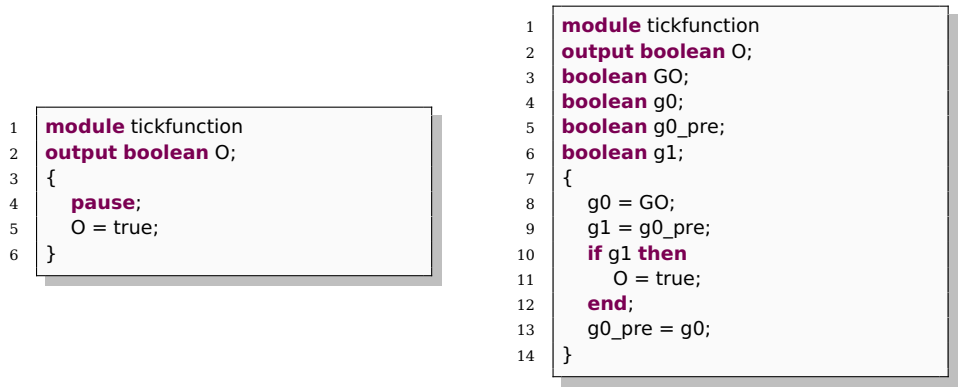
1. Ausdrücke und Variablenzuweisungen sind typkorrekt.
2. Es sind weder überflüssige noch fehlende Programmlabels vorhanden.
3. Es sind keine goto-Sprünge in parallel oder aus parallelen Regionen vorhanden.

Ein SCChart besitzt Ein-, Ausgabe- und lokale Variablen, um das reaktive Verhalten zu beschreiben. Da SCCharts mit SCL modelliert werden können, können in einem SCChart ebenso Ein- und Ausgabevariablen wie lokale Variablen deklariert werden. In Abbildung 4.6a ist ein SCL-Programm gezeigt, welches einige der erwähnten Befehle und eine Deklaration einer Ausgabevariablen beinhaltet. Das Programm setzt den Ausgang O nach einer pause, somit im zweiten Tick, auf true.

Sequentielles SCL

Sequential SCL (*sequentielles SCL*) ist eine Sprache, die auf einer Teilmenge von SCL beruht. Im sequentiellen SCL gibt es weder pause noch parallele Anweisungen [Smy13]. Jedes ausführbare SCL Programm kann in ein sequentielles SCL-Programm ohne Nebenläufigkeit mit äquivalentem Verhalten übersetzt werden.

Ein sequentielles SCL Programm ist um sogenannte *Guards* angereichert worden. Bei Guards handelt es sich um zusätzlich eingeführte Variablen, die den Kontrollfluss des Programms steuern. So muss ein Guard für eine Anweisung aktiv auf true gesetzt sein, damit diese ausgeführt wird. Ein sequentielles SCL-Programm spiegelt eine Funktion wider, die in jedem Tick vollständig berechnet wird: die sogenannte Tick-Funktion. In jedem Tick berechnet die Funktion in Abhängigkeit des Zustandes, in dem sich das Programm befindet, die Ausgaben.



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

(a) Einfaches SCL-Programm

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

(b) Einfaches SCL-Program in sequentiellen SCL

Abbildung 4.6. Transformationsbeispiel von SCL zu sequentiellen SCL

Da pause-Anweisungen nicht mehr vorhanden sind, muss der Kontrollfluss in weiteren Registern gespeichert werden. So werden Guards, die nach einer pause aktiv sein sollen, auf den Guard der pause des vorherigen Ticks gesetzt. Werte aus vorherigen Ticks werden dabei in sogenannten *pre*-Variablen gespeichert. Diese werden am Ende der Tick-Funktion zugewiesen.

Ein Beispiel eines sequentiellen SCL-Programms ist in Abbildung 4.6b zu sehen. Transformiert wurde das SCL-Programm aus Abbildung 4.6a. Im ersten Tick wird die pause ausgeführt. Das bedeutet, dass das Guard g1 in diesem Tick nicht aktiv (false) ist. g0 wird durch die GO-Variable gesetzt. Die GO-Variable startet die Ausführung eines SCL-Programms. Am Ende des ersten Ticks wird g0_pre auf den aktuellen Wert von g0 gesetzt. Im zweiten Tick wird das Guard g1 auf den Wert der pause aus dem vorherigen Tick (g0_pre) gesetzt. g1 ist nun aktiv (true) und O wird gesetzt.

4.2 SCCharts Transformation

Die ersten Schritte der Transformation von SCCharts nach Hardware bestehen daraus, Core SCCharts in SCL und danach in sequentielles SCL zu übersetzten. Wie zu Beginn in Kapitel 4 bereits erwähnt, sind diese Transformationen Teil der Diplomarbeit von Herrn Smyth. Da diese Transformationen in dieser Arbeit verwendet werden, folgt in diesem Kapitel eine kurze Vorstellung. Detailliertere Informationen, sowie Implementierungseigenschaften sind in der Arbeit “Code Generation for Sequential Constructiveness” von Smyth zu finden [Smy13].

4.2.1 Extended SCCharts zu Core SCCharts Transformation

Im ersten Schritt der Transformation müssen extended SCCharts in Core SCCharts übersetzt werden. Wie bereits im Kapital 4.1.2 beschrieben, haben extended SCCharts eine angereicherte, auf Core SCCharts basierende Syntax, welche es dem Modellierer erlaubt, mit zusätzlichen Elementen komplexere SCCharts kompakter zu modellieren.

Jedes der syntaktisch komplexeren Elemente lässt sich mit einer oder mehreren M2M Transformationen in eine äquivalente Core Variante überführen [vHMA+13a]. Betreffende Elemente werden in einem extended SCChart gesucht und durch äquivalente Elemente ersetzt. Dabei muss es sich nicht zwangsläufig direkt um Core Elemente handeln. In der Regel müssen mehrere aufeinanderfolgende Transformationen durchgeführt werden, bevor ein extended SCChart komplett umgeformt wurde.

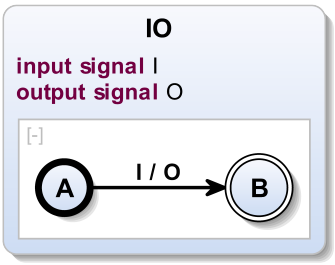
Ein Beispiel, welches diese Art der Transformation verdeutlicht, ist in Abbildung 4.7 dargestellt. Das SCChart aus Abbildung 4.7a ist ein extended SCChart und wird in den folgenden Schritten in ein Core SCChart übersetzt. Signale gehören nicht zu den Core Elementen. Sie können durch boolsche Variablen ersetzt werden (Abbildung 4.7b), so wird aus signal O, output boolean O. Signale werden in jedem Tick, aufgrund des Signalkohärenz Gesetztes (Abschnitt 1.1.1, auf dem sie beruhen, zurückgesetzt. Das Zurücksetzen wird durch *During Actions* erreicht. Bei *During Actions* handelt es sich ebenfalls um extended Elemente. Das vorliegende SCChart muss in einer weiteren Transformation übersetzt werden.

Im nächsten Schritt müssen die *During Actions* transformiert werden. Sie können durch eine instantane Transition ersetzt werden, die die Variablen zurücksetzt. Das so entstandene SCChart (Abbildung 4.7c) enthält nur noch Core Elemente, womit es die Anforderungen an ein gültigen Core SCChart erfüllt.

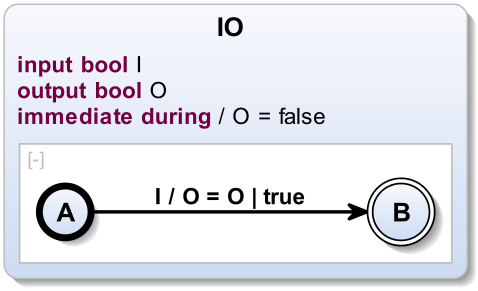
4.2.2 Core SCCharts zu SCL Transformation

Da SCCharts und SCL auf dem selben Ausführungsmodell beruhen, haben sie ähnliche Programmierkonstrukte. Für SCCharts Elemente gibt es eine entsprechende Repräsentation in SCL. So werden zum Beispiel nebenläufige Regionen in SCCharts in ein fork par join Konstrukt in SCL übersetzt. Die in der Diplomarbeit “Code Generation for Sequential Constructiveness” von Smyth [Smy13] vorgestellte Transformation von SCCharts zu SCL erfolgt in zwei Schritten. Zuerst wird eine naive Transformation ausgeführt, bei der unerwünschte Artefakte entstehen können, diese werden in einem weiteren Schritt beseitigt und optimiert.

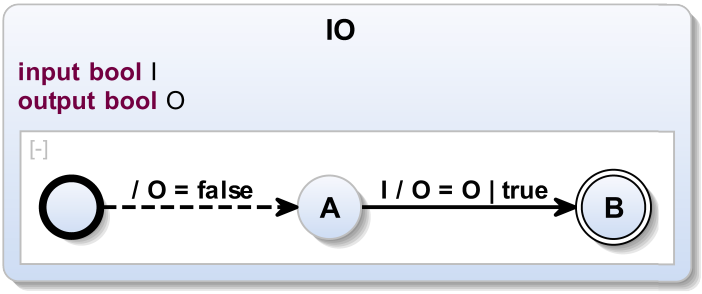
Im ersten Schritt werden die SCCharts nach SCL transformiert. Jedes SCChart enthält einen *Main state*, welcher eine oder mehrere Regionen enthält. Dieser *Main state* wird mit seinen Regionen und Zuständen rekursiv durchlaufen und dabei Schritt für Schritt transformiert. Jedes SCCharts Element hat ein unterscheidbares textuelles Quellcode-Element in SCL. So wird für jeden Zustand, der während der Rekursion



(a) Extended SCChart



(b) Extended SCCharts mit During Actions



(c) Core SCChart

Abbildung 4.7. Beispiel einer Transformation von Extended SCCharts zu Core SCCharts [MSvHM13]

durchlaufen wird, ein entsprechendes Quellcode-Element erzeugt. Die Abbildung 4.8 gibt einen Überblick über die einzelnen Schritte der Zustands-Transformation.

Transitionen können annähernd direkt in SCL übersetzt werden. Jeder *Trigger* wird in ein Conditional übersetzt und jeder *Effect* in einen entsprechenden Ausdruck. Die SCL-Zuweisung wird ausgeführt, wenn die Bedingung zu `true` ausgewertet wird. Sofern kein *Trigger* vorhanden ist, handelt es sich um eine *default*-Transition, welche ausgeführt wird, wenn kein anderer *Trigger* gültig ist. Solche default-Transition von einem zu einem anderen Zustand wird durch einen `goto`-Sprung ersetzt.

Im zweiten Schritt der Transformation werden Optimierungen [Smy13] vorgenommen. Bei der naiven Transformation entstehen Artefakte, die nicht benötigt werden. Sie können aus dem SCL-Quellcode entfernt werden, ohne die Semantik des Programms zu ändern. So können bei der Transformation `goto`-Sprünge zu entsprechenden *Labels* entstehen, welche in aufeinanderfolgenden Zeilen stehen. Solche Artefakte können mit der *Goto Optimization* und der *Label Optimization* entfernt werden. Weitere Optimierungen wie *Self-loop Optimization*, *State Ordering Optimization*, *Duplicate Transition Optimization* und *WTO Transition Optimization* werden durchgeführt, um den Quellcode weiter zu komprimieren und zu optimieren.

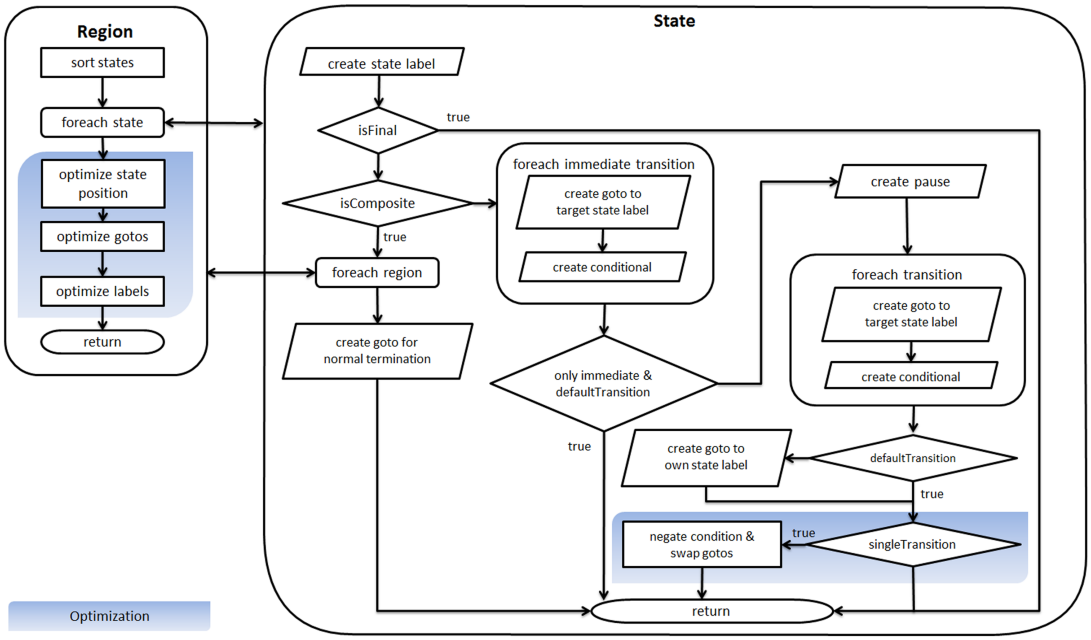


Abbildung 4.8. Core SCChart zu SCL — Transformationsübersicht für Zustände [Smy13]

In dem Listing 4.1 ist ein transformiertes *ABO*-Beispiel zu sehen. Es wurden die beiden Transformationsschritte auf das Modell aus Abbildung 4.4 angewendet.

Zu Beginn des SCL-Quellcodes sind die Deklarationen der Ein- und Ausgabevariablen sowie der lokalen Variablen, zu sehen. Die beiden nebenläufigen Regionen HandleA und HandleB wurde in zwei Threads __WaitAB_HandleA_WaitA und __WaitAB_HandleB_WaitB übersetzt. Die verwendeten Transitionen wurden in die entsprechenden Bedingungen und Anweisungen übersetzt.

4.2.3 Der SC-Graph

Wie bereits in Abschnitt 4.1.3 erwähnt, gibt es für jede SCL-Anweisung ein entsprechende grafische Repräsentation, den Sequentially Constructive Graph (SCG). Die pause-Anweisung wird mit zwei Knoten dargestellt, wo hingegen die anderen Anweisungen einzelne grafische Komponenten besitzen. Ein SCL-Programm kann somit auch in visualisierter Form dargestellt werden, in einem sogenannten SC-Graph oder kurz *SCG*. Auf der Ebene der grafischen Präsentation können verschiedene Analysen durchgeführt werden, die unter anderem Schedulingprobleme ausfindig machen können. Die pause-Anweisung ist in *Surface* und *Depth* aufgeteilt. Ist der Kontrollfluss in der *Surface*, endet der Kontrollfluss an dieser Stelle für den aktuellen Tick. Im nächsten Tick wird der Kontrollfluss in der *Depth* fortgesetzt. Eine grafische Repräsentation von *ABO* ist in Abbildung 4.9 dargestellt.

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

Listing 4.1. ABO — mit optimiertem SCL Code Code

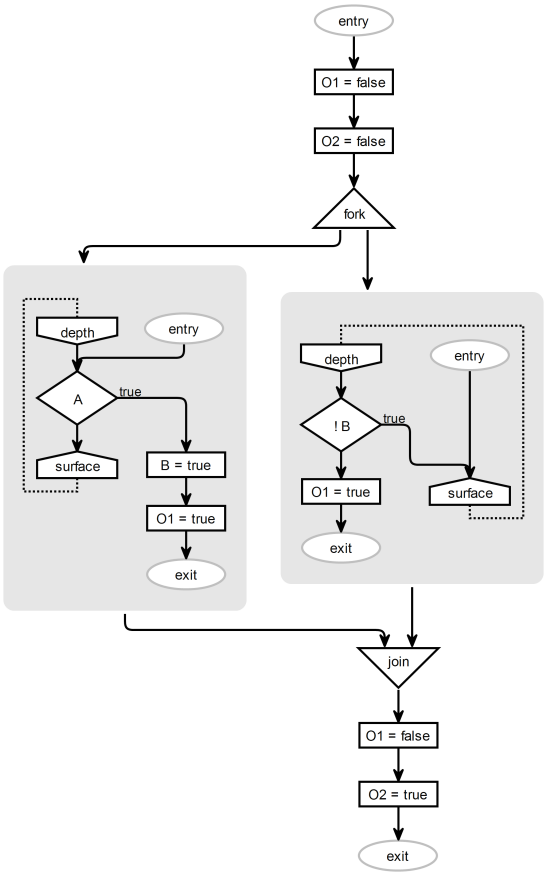


Abbildung 4.9. ABO als SCG visualisiert

4.2.4 Analysen auf dem SCG

Im Folgenden werden zwei Analysen des SCL-Programms von Smyth [Smy13] erläutert. Diese Analysen werden auf ein SCL-Programm angewendet, um bestimmen zu können, ob für das Programm ein statisches Schedule existiert und um das Programm später in eine sequentielle SCL Variante überführen zu können.

Abhängigkeitsanalysen

Um zu überprüfen, ob es für ein SCL-Programm ein gültiges Schedule gibt, müssen die nebenläufigen Abhängigkeiten von Variablen eines Programms ermittelt werden. So ist ein SCL-Programm nicht ausführbar, wenn es unlösbare Konflikte beim Zugriff auf geteilte Variablen gibt. Aus diesem Grund müssen Abhängigkeiten beim nebenläu-

figen Variablenzugriff identifiziert und zugeordnet werden. Die Zugriffe auf Variablen werden in lesende, absolut schreibende und relativ schreibende Zugriffe unterteilt. Absolute Schreibzugriffe sind von der Form $O = true$ und relative Schreibzugriffe sind von der Form $x = f(x, ex)$, zum Beispiel $O = O||true$. So können die folgenden Abhängigkeiten zwischen Threads bei nebenläufigen Variablenzugriffen entstehen:

- ▻ absoluter Schreibzugriff - absoluter Schreibzugriff
- ▻ absoluter Schreibzugriff - relativer Schreibzugriff
- ▻ absoluter Schreibzugriff - lesender Zugriff
- ▻ relativer Schreibzugriff - lesender Zugriff

Auf Basis dieser Analyse kann später getestet werden, ob die Zugriffe in eine konfliktfreie Reihenfolge gebracht werden können und somit ein statisches Schedule für das betreffende Programm existiert. In Abbildung 4.10 ist eine Abhängigkeitskante eingezeichnet, welche einer absoluten Schreibzugriff-Lesezugriff-Abhängigkeit entspricht. Der eine Thread schreibt auf die Variable B (absoluter Schreibzugriff), während der andere Thread den Wert aus B lesen möchte. Für ein gültiges statisches Schedule, muss der Schreibzugriff vor dem Lesezugriff ausgeführt werden.

Basic Block Analyse

Basic Blocks (BBs) [Smy13] sind zusammengefasste SC-Anweisungen, die ausgeführt werden können, ohne dass das Programm den aktuellen Kontrollfluss verlassen muss. Es sind Anweisungen, die zusammenhängend ausgeführt werden können. Im schlechtesten Fall kann für jede SCL-Anweisung ein einzelner BB verwendet werden, jedoch ist erwünscht, möglichst viele Anweisungen in einem Block zusammenzufassen. Die BB-Generierung unterliegt folgenden Regeln:

- ▻ Ein BB beginnt mit der ersten Anweisung in einem Thread oder wenn eine Anweisungen in der SCG-Repräsentation zwei oder mehr eingehende Kontrollflusskante hat.
- ▻ Ein BB endet mit einer fork-Anweisung, die folglich zwei oder mehr ausgehende Kanten hat.
- ▻ Eine pause-Anweisung wird grafisch mit zwei Knoten dargestellt. An einer pause werden die BB aufgespalten.
- ▻ An einer join-Anweisung beginnt ein neuer BB.
- ▻ Jede SCL-Anweisung kann nur zu einem BB gehören.

Ein Basic Block kann innerhalb eines Ticks *aktiv* oder *inaktiv* sein. Ein aktiver Zustand eines BB leitet sich von der Ausführung der vorhergehenden Blöcke ab. Man spricht von sogenannten *guarded BB*. Der Guard des ersten Blockes innerhalb

eines Programms ist von der GO-Variablen abhängig. GO wird normalerweise bei der Initialisierung oder beim Reset des Programms einmalig gesetzt. Der ausgehende Kontrollfluss eines BB kann als Aktivator für folgende Blöcke dienen. Eine SCL-Anweisung wird ausgeführt, wenn der dazugehörige BB aktiv ist.

Die visualisierten BB sind exemplarisch in Abbildung 4.10 gezeigt. Die lila gefärbten Kästen um die SCL-Anweisungen sind BB. An der oberen linken Ecke ist der Name des BBs aufgeführt. So werden zum Beispiel in BB g3 zwei Anweisungen vereint. Am linken Rand der BB stehen die Blöcke, von denen der aktuelle BB abhängig ist. So kann g6 ausgeführt werden, wenn g0 aktiv ist oder g8 und das Conditional zu true ausgewertet wird.

4.2.5 SCL zu sequentiellem SCL Transformation

Sofern ein statisches Schedule gefunden werden kann, welches den SC-Kriterien entspricht (unter anderem Abschnitt 4.1.1), kann es in ein sequentielles SCL-Programm überführt werden. Können die Abhängigkeiten zwischen Variablenzugriffen nicht aufgelöst und kein statisches Schedule gefunden werden, wird das Programm abgelehnt. Wie in Abschnitt 4.1.3 beschrieben, gibt es in sequentiellen SCL-Programmen keine Nebenläufigkeit und keine pause-Anweisungen mehr. Der generierte Quellcode entspricht der erwähnten Tick-Funktion, welche je nach Zustand des Programms die Ausgaben berechnet.

In die Generierung eines sequentiellen SCL-Programms fließen die Ergebnisse der Analysen mit ein. So wird das Programm entsprechend der gefunden Abhängigkeiten in eine topologische Sortierung gebracht, dem statischen Schedule. Die erzeugten BB, werden verwendet, um den Kontrollfluss des Programms zu steuern. So wird unter anderem für einen BB, der eine Zuweisung enthält, ein Conditional erzeugt, welches als Bedingung den Guard des BBs und im *then*-Zweig die Zuweisung hat.

Da ein sequentielles SCL-Programm keine pause-Anweisungen mehr enthält, jedoch durchaus die Semantik einer pause erhalten bleiben muss, wurden spezielle pause-Variablen eingeführt. Die *Surface* einer Pause wird zu einer normalen Guard-Variable. Die *Depth* hingegen wird zu einer pre-Variablen, die zur passenden Guard-Variablen gehört. Am Ende der Tick-Funktion werden den *pre*-Variablen der Wert aus der Guard-Variablen zugewiesen. Im nächsten Tick führen die *pre*-Variablen somit die Information, ob die Pause (*Surface*) im vorhergehenden Tick aktiv war und steuert, ob der Kontrollfluss nach der Pause (*Depth*) fortgesetzt werden kann. In Abbildung 4.6 ist ein einfaches Beispiel einer pause-Anweisung in sequentiellen SCL abgebildet, dabei entsprechen die Variablen g_0 und $g0\_pre$ der pause-Anweisung.

In Abbildung 4.2 ist ABO als sequentielles SCL-Programm aufgeführt. Der komplexere Ausdruck, der das Guard $g1$ berechnet, entspricht der *join*-Anweisung zweier Threads und ist angelehnt an Berrys *Synchronizer* [Ber02, Smy13].

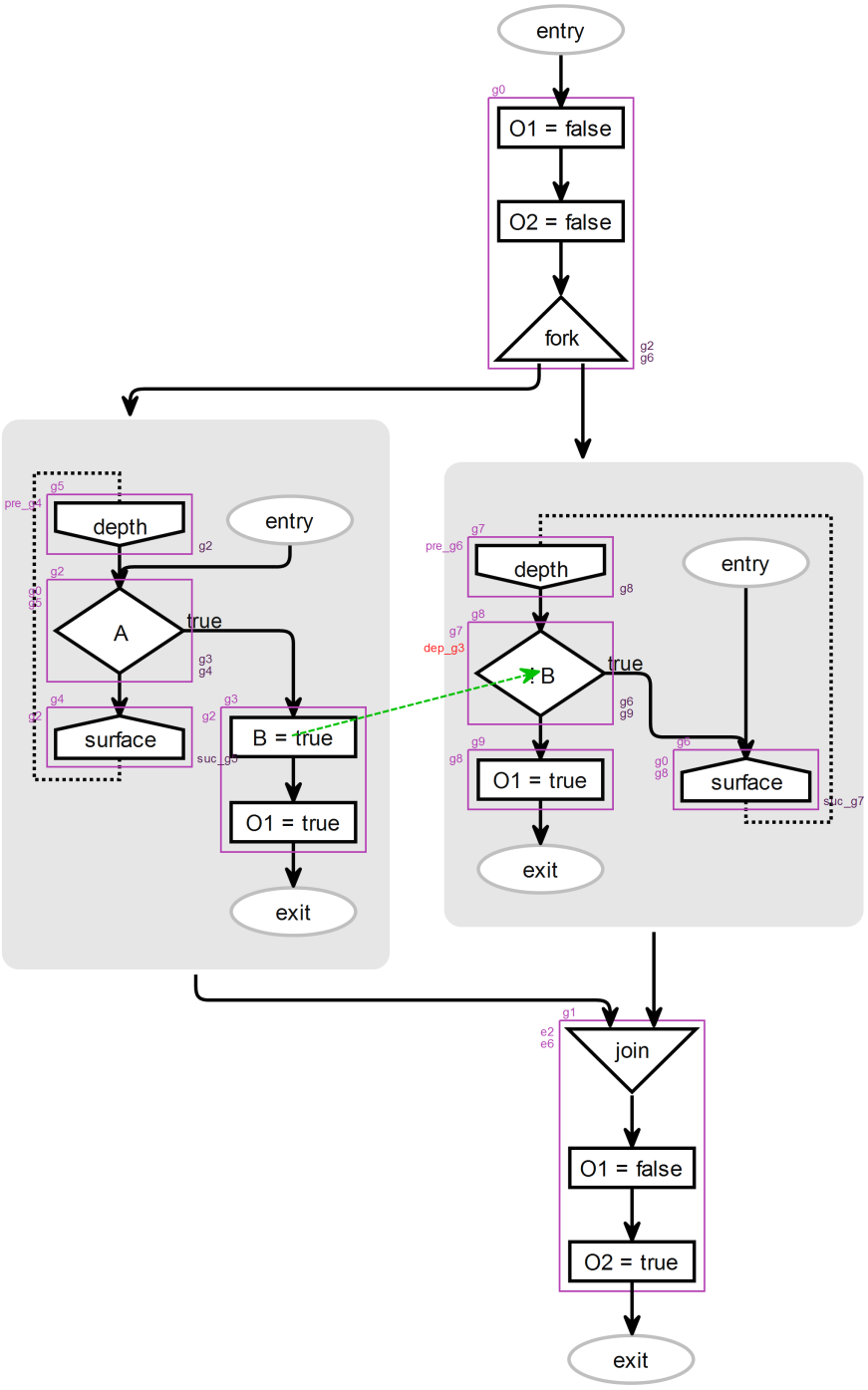


Abbildung 4.10. ABO als SCG mit Basic Blocks und Abhängigkeitskanten

```
1  module ABO_tick
2  input output boolean A, B;
3  output boolean O1, O2;
4  boolean GO, g0, g1, e2, e6, g2, g3;
5  boolean g4, g5, g6, g7, g8, g9;
6  boolean g4_pre, g6_pre;
7  {
8     g0 = GO;
9     if g0 then
10       O1 = false;
11       O2 = false;
12    end;
13    g5 = g4_pre;
14    g7 = g6_pre;
15    g2 = g0 || g5;
16    g3 = g2 && A;
17    if g3 then
18       B = true;
19       O1 = true;
20    end;
21    g4 = g2 && ! A;
22    g8 = g7;
23    g9 = g8 && B;
24    if g9 then
25       O1 = true;
26    end;
27    g6 = g0 || ( g8 && ! B );
28    e2 = ! ( g4 );
29    e6 = ! ( g6 );
30    g1 = ( g3 || e2 ) && ( g9 || e6 ) && ( g3 || g9 );
31    if g1 then
32       O1 = false;
33       O2 = true;
34    end;
35    g4_pre = g4;
36    g6_pre = g6;
37 }
```

Listing 4.2. ABO — sequentieller SCL Code

4.3 Naive VHDL Transformation

Die erste Transformation, die in dieser Arbeit vorgestellt wird, verfolgt einen naiven Ansatz, bei dem sequentielle SCL-Programme ohne mögliche Optimierungen annähernd direkt in eine Hardwarebeschreibung übersetzt werden. Das SCL-Modell wird in eine VHDL-Beschreibung übersetzt, die eine Beschreibung des Verhaltens und eine Interfacebeschreibung beinhaltet, die sogenannte *Entity*.

Bei der Hardware-Programmierung werden alle Anweisungen, im Gegensatz zur herkömmlichen Programmierung, parallel ausgeführt. Ein kleines Beispiel soll ein entstehendes Problem verdeutlichen. So können die Zuweisungen, wie im Listing 4.3 dargestellt, in einer Sprache wie C problemlos ausgeführt werden. Am Ende der Ausführung hat die Variable i den Wert true, da beide Anweisungen nacheinander ausgeführt wurden.

```
1 i = false;
2 i = true;
```

Listing 4.3. Sequentielle Zuweisungen zu einer Variablen i

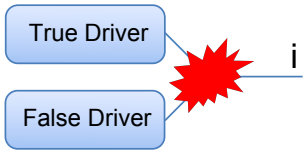


Abbildung 4.11. Ein Kurzschluss in Hardware durch zwei parallele Zuweisungen zu einem Signal

In VHDL werden diese Zuweisungen in einen Schaltkreis übersetzt. Schaltkreise arbeiten per se parallel. Die Variable wird in VHDL zu einer Leitung, die eine Spannung führt, übersetzt. Für die beiden Zuweisungen werden Treiber für diese Leitung generiert. Angenommen ein true entspricht dem Spannungswert *5 Volt* und ein false der Spannung *0 Volt*. Die beiden Treiber würden ihren Spannungswert parallel auf die Leitung schreiben, was zu einen Kurzschluss führen würde, siehe Abbildung 4.11.

Bei der Hardware-Programmierung müssen sequentielle Programme deshalb gesondert betrachtet werden. VHDL bietet das process-Konstrukt (Prozess) an, mit dem auch in Hardwarebeschreibungen sequentiell programmiert werden kann. Hier berechnet der ISE Compiler einen Schaltkreis der die sequentielle Ausführung abbildet.

In diesem Lösungsansatz wird der sequentielle SCL-Quellcode zu ein VHDL-Prozess transformiert. Zusammen mit einer Interface-Deklaration aus dem SCL-Modell kann das Modell in Hardware übersetzt werden.

4.3.1 Das VHDL Prozess Konstrukt

Der VHDL-Prozess ist das Kernkonstrukt in dieser Transformation. Wie oben erwähnt, kann mit Hilfe des Prozesses sequentiell, wie C oder Java, programmiert werden. Eine detailliertere Erklärung zu Prozessen ist in Abschnitt 4.4.3 zu finden. Sie ist für das Verständnis dieser Transformation nicht zwingend erforderlich.

Ausgangspunkt für diese Transformation ist der sequentielle SCL-Programmcode. Wie der Name bereits verrät, handelt es sich um sequentiellen Quellcode ohne nebenläufige Regionen. Die Idee ist es, diesen sequentiellen Quellcode in einen VHDL-Prozess zu transformieren. Das sequentielle SCL-Programm kann mit der Tick-Funktion gleichgesetzt werden. Eine Tick-Funktion wird in Software von einer Kontrollinstanz zu jedem Tick ausgeführt. Innerhalb von VHDL wird das erreicht, indem die Berechnung des Prozesses in jedem Tick gestartet wird. Dazu besitzen Prozesse die Möglichkeit, solche “Startsignale” gesondert zu definieren. In dem Beispiel in Abbildung 4.12 ist die Übersetzung eines sequentiellen SCL-Programms zum VHDL-Prozess gezeigt.

Der Prozess wird mit dem Schlüsselwort process und einem optionalen Namen deklariert. Darauf folgt die Definition von Variablen. Das tick-Signal startet die

```
module tickfunction
output boolean O;
boolean GO;
boolean g0;
boolean g0_pre;
boolean g1;
{
   g0 = GO;
   g1 = g0_pre;
   if g1 then
      O = true;
   end;
   g0_pre = g0;
}
```

(a) Einfaches sequentielles SCL Programm

```
p: process

-- local in/out Variables
variable O_local : boolean := false;
-- local variables
variable GO_local : boolean := false;
variable g0 : boolean := false;
variable g0_pre : boolean := false;
variable g1 : boolean := false;

begin

    wait until rising_edge(tick);
    if(reset = true) then
        O_local := false;
        GO_local := false;
        g0 := false;
        g0_pre := false;
        g1 := false;
    else
        GO_local = GO;

        -- main program
        g0 := GO_local;
        g1 := g0_pre;
        if (g1) then
            O_local := true;
        end if;
        g0_pre := g0;

        -- set outputs
        O <= O_local;
    end if;
end process p;
```

(b) VHDL Process des einfachen SCL Programms

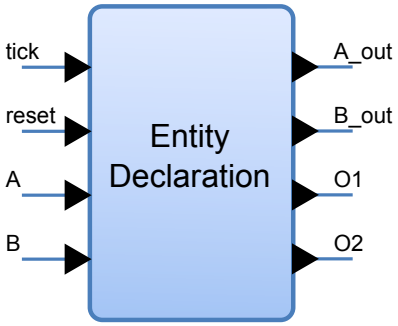
Abbildung 4.12. Transformation von sequentiellem SCL zu einem VHDL Prozess

Berechnung innerhalb des Prozesses mit jeder steigenden Flanke (wait until rising_edge(tick)). Nach der begin-Anweisung folgt der Quellcode der Tick-Funktion. Der Prozess ist noch um das Rücksetzten der Variablen im Falle eines Resets erweitert. Das GO-Signal ist ein Signal außerhalb des Prozesses, das die Berechnung initial startet.

Ein Prozess ermöglicht nicht nur die sequentielle Programmierung, sondern er verhält sich auch wie ein sequentielles Programm. Damit Werte in Hardware gespeichert werden, um sich zum Beispiel Zustände zu merken, müssen Register verwendet werden. In einem Prozess werden automatisch Register angelegt für Werte, die gespeichert werden müssen. Der Entwickler muss keine Register explizit anlegen. Obwohl eine Variable ein persistentes Verhalten besitzt, wird vom Compiler nicht für jede Variable ein Register angelegt. Eine Variable wird als Register angelegt,

```
1 input output boolean A, B;
2 output boolean O1, O2;
3 boolean GO;
4 boolean g0, g1, e2, e6, g2, g3;
5 boolean g4, g5, g6, g7, g8, g9;
6 boolean g4_pre, g6_pre;
```

(a) Deklaration im sequentiellen SCL Code — ABO



(b) Übersicht der Entity Deklarierung als Blockschaltbild

Abbildung 4.13. Beispiel einer Entity Deklaration in SCL und dem passenden VHDL-Blockschaltbild für ABO

wenn der Wert der Variablen gelesen wird, bevor er geschrieben wird. Die anderen Variablen werden in Signale transformiert.

4.3.2 Interface Deklaration

Jede VHDL-Beschreibung basiert auf den Sichten wie im Abschnitt 3.4.1 dargestellt. Die externe Sicht beschreibt die Schnittstellendefinition eines Schaltkreises. Die Schnittstellendefinition legt die Ein- bzw. Ausgänge fest, die der Schaltkreis, die Entity, zur Kommunikation zur Verfügung stellt. Die benötigten Ein- und Ausgänge lassen sich aus der Definitionsbeschreibung des sequentiellen SCL-Programms entnehmen, siehe Listing 4.13a.

Eingaben (inputs) und Ausgaben (outputs) der SCL-Definition werden zu Ein- und Ausgängen der Entity transformiert. Eine Besonderheit haben die Ein-Ausgabevariablen (input output) des SCL-Modells. Sie werden in einzelne Ein- und Ausgabeleitungen aufgetrennt. Diese Designentscheidung wird im Abschnitt 4.4.3 näher begründet. Der Name des Eingangssignals stimmt mit dem Variablennamen überein. Das Ausgangssignal bekommt die Erweiterung $\_out$. In Abbildung 4.13b ist die externe Sicht von ABO gezeigt. Die Eingänge tick und reset sind Steuereingänge für den Schaltkreis. Über den tick-Eingang wird die Berechnung der Tick-Funktion gestartet und der reset-Eingang dient zum Zurücksetzen des Schaltkreises.

4.3.3 Sequentielles SCL zu VHDL Transformation

Die naive Transformation soll in erste Linie zeigen, dass es relativ einfach ist, Hardware aus SCCharts zu synthetisieren.

Wie zu Beginn erwähnt, handelt es sich um eine einfache Transformation, die geradezu direkt VHDL-Code aus SCCharts generieren kann. Die Transformation bekommt ein SCChart als Eingabe, welches transformiert werden soll. Zu Beginn wird die Entity Deklaration aus der SCL-Deklaration erstellt. Die benötigten Variablen, die ebenfalls der SCL-Deklaration entnommen werden können, werden in VHDL-Variablen transformiert. Im weiteren Verlauf wird der SCL-Quellcode, unter Berücksichtigung der Syntax, nach VHDL transformiert. So müssen zum Beispiel logische Operatoren wie $\&\&$ und $||$ in SCL zu and und or in VHDL übersetzt werden. Weiterhin werden während der Transformation Steuersignale erzeugt. Dazu gehören des Tick-Signal tick und das Resetsignal reset. Ersteres führt den Schaltkreis in jedem Tick aus, während Letzteres bei einem Reset die Variablen zurücksetzt. Im folgenden Abschnitt sollen die einzelnen Schritte der Synthese beispielhaft am ABO SCChart illustriert werden.

4.3.4 Das ABO Beispiel

Das “Hello World” des sequentiell konstruktiven Sprachen ist das ABO-Beispiel. Im Folgenden wurde aus dem ABO-Beispiel, aus Abbildung 4.4, automatisch sequentieller SCL-Code generiert, Abbildung 4.2. Dieser Quellcode wurde mit der vorgestellten Transformation in VHDL übersetzt, Abbildung 4.14 zeigt den transformierten Prozess. In Abbildung 4.15 ist der synthetisierte Schaltkreis zu sehen.

In dieser Arbeit stellt diese naive Transformation eine untergeordnete Rolle dar, aus diesem Grund wurde nur ein Überblick über die Transformation gegeben. Als Nächstes wird ein weiterer Ansatz für eine Transformation von SCCharts zu Hardware gezeigt.

4.4 Zweiter Transformationsansatz

In diesem Kapitel wird ein zweiter Ansatz für die Transformation von SCCharts zu Hardware vorgestellt. Der zweite Lösungsansatz behebt das Problem, dass für eine Leitung mehr als ein Treiber existiert, dass bei der Übersetzung von sequentiellen Code in VHDL entsteht, siehe Unterkapitel 4.3, auf eine andere Weise. In der ersten Lösung übernimmt das Prozess-Konstrukt die sequentielle Ausführung des Codes und der ISE-Compiler löst die Datenabhängigkeiten auf und generiert daraus einen entsprechenden Schaltkreis.

In der nun vorgestellten Transformation werden die Datenabhängigkeiten in einem weiteren Zwischenschritt aufgelöst, sodass der Code nach VHDL übersetzt werden kann. Dafür wird der sequentielle SCL-Code in eine Static Single Assignment (SSA)-Form (SSA SCL) übersetzt. Diese wird wiederum nach VHDL übersetzt. In diesem Kapitel wird zu Beginn das SSA vorgestellt und einige Besonderheiten, die bei der Transformation zu beachten sind. Ebenso wird die entwickelte *One-Pass-*

```vhdl
ENTITY ABO IS
PORT(
    -- control
     tick : IN std_logic;
     reset : IN boolean;
     -- inputs
     A: IN boolean;
     B: IN boolean;
     -- outputs
     A_out : OUT boolean;
     B_out : OUT boolean;
     O1 : OUT boolean;
     O2 : OUT boolean
    );
END ABO;

ARCHITECTURE behavior OF ABO IS
begin
p: process
-- *** Variable Declaration ***
begin
    wait until rising_edge(tick);
    if(reset = true) then
    -- *** Variable Reset ***
       A_local := false;
      B_local := false;
      A_out_local := false;
      B_out_local := false;
      O1_local := false;
      O2_local := false;
      g0 := false;
      g1 := false;
      e2 := false;
      e6 := false;
      g2 := false;
      g3 := false;
      g4 := false;
      g4_pre := false;
      g5 := false;
      g6 := false;
      g6_pre := false;
      g7 := false;
      g8 := false;
      g9 := false;
    else
      -- update local variables
      A_local := A;
      B_local := B;

      -- main program
      g0 := GO_local;
      if (g0) then
          O1_local := false;
          O2_local := false;
      end if;
      g5 := g4_pre;
      g7 := g6_pre;
      g2 := g0 or g5;
      g3 := g2 and A_local;
      if (g3) then
          B_local := true;
          O1_local := true;
      end if;
      g4 := g2 and not A_local;
      g8 := g7;
      g9 := g8 and B_local;
      if (g9) then
          O1_local := true;
      end if;
      g6 := g0 or (g8 and not B_local);
      e2 := not (g4);
      e6 := not (g6);
      g1 := (g3 or e2) and (g9 or e6) and (g3 or g9);
      if (g1) then
          O1_local := false;
          O2_local := true;
      end if;
      g4_pre := g4;
      g6_pre := g6;

      -- set outputs
      A_out <= A_local;
      B_out <= B_local;
      O1 <= O1_local;
      O2 <= O2_local;
   end if;
end process p;
-- GO generation omitted
end behavior;
```

Abbildung 4.14. Automatisch generierter VHDL Code von ABO mit der naiven Transformation

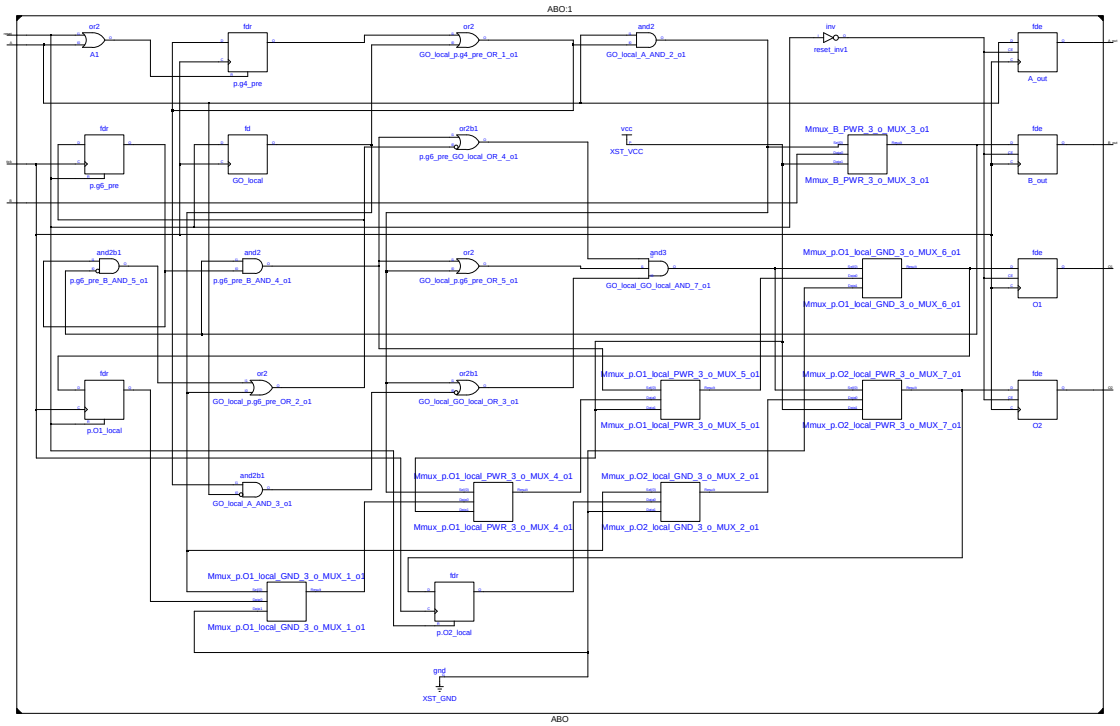


Abbildung 4.15. Synthetisierter Schaltkreis von ABO aus dem mit der naiven Transformation transformierten VHDL Code

Transformation erklärt. Im zweiten Teil wird eine Transformation vorgestellt, die aus dem SSA-transformierten sequentiellen SCL-Code (SSA SCL Code) VHDL erzeugt.

4.4.1 Das Static Single Assignment

Die Static Single Assignment (SSA)-Form ist eine Zwischensprache, die bei Compilern verwendet wird, um Optimierungen vorzunehmen und effizienten Code für imperative Sprachen zu erzeugen. SSA wurde von Wegman, Zadeck, Alpern und Rosen [App98, AWZ88, RWZ88] für effiziente Berechnung von Datenflussproblemen wie zum Beispiel global Variablennummerierung, Deadcode Eliminierung und Konstanten Propagation entwickelt. Hier wird zu Beginn die Idee vom SSA informell vermittelt, bevor die Idee des Algorithmus erklärt wird.

Zuweisungen

SSA ist eine Datenflussanalyse und wird in der Regel auf Kontrollflussgraphen angewendet. Zu Beginn wird eine einfache Zuweisungsreihenfolge unter der SSA-Form betrachtet, Abbildung 4.16a. Dabei wird jeder linken Variablen v, einer Zuweisung eine neue, eindeutig bezeichnete Instanz eingerichtet. Der Name entspricht einer

```
1 v = 0;
2 x = v + 1;
3 v = 2;
4 y = v + 3;
```

(a) Einfache Zuweisungen zu veschiedenen Variablen

```
1 v_1 = 0;
2 x_1 = v_1 + 1;
3 v_2 = 2;
4 y_1 = v_2 +3;
```

(b) Einfache Zuweisungen zu verschiedenen Variablen in der SSA Form

Abbildung 4.16. Von einer einfachen Variablen Zuweisungen zu Variablen Zuweisungen in der SSA-Form

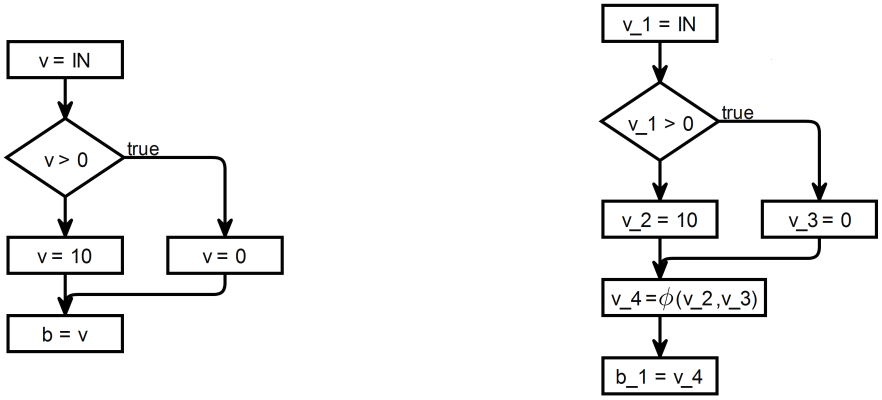
Versionierung der Variablen, sie erhält eine Versionsnummer i. Die Variable hat folgenden eindeutigen Namen: v_i. Jede Variable wird so nur einmal zugewiesen. Bei lesenden Zugriffen auf Variablen werden die Variablen durch ihre Instanz mit der derzeitig höchsten Version an der Programmposition ersetzt, siehe Abbildung 4.16b.

Allgemein wird in jeder Zuweisung der Form $v = e$, wobei v eine Variable und e ein Ausdruck ist, der v nicht referenziert, die Variable v durch eine Zuweisung mit einer versionierten Variable v_i ersetzt, $v_i = e$. Wird eine Variable x gelesen, $v = x$, so wird x durch die aktuelle Instanz mit dem höchsten Index von x ersetzt, $v_i = x_k$, i und k sind Versionsnummern der jeweiligen Variablen.

Kontrollflussverzweigungen

In Programmen kommen nicht nur einfache Zuweisungsfolgen vor. So werden auch Programmierkonstrukte wie *bedingte Sprünge* oder *Schleifen* verwendet. Solche Konstrukte beinhalten eine Verzweigung des Kontrollflusses. An Kontrollflussverzweigungen können verschiedene Versionen von Variablen entstehen. Im Folgenden wird eine Kontrollflussverzweigung mit zwei Kontrollflüssen betrachtet, wobei die beiden Kontrollflüsse mit linker und rechter Kontrollfluss bezeichnet werden. Da während der Analyse nicht unbedingt entschieden werden kann, welcher Kontrollfluss genommen wird, muss beim Zusammentreffen der beiden Kontrollflüsse herausgefunden werden, welche Version die Aktuellste ist. Die verschiedenen Versionswerte müssen zu einer neuen Variable zusammengeführt werden, bevor die Variable erneut gelesen werden kann. ϕ-Funktionen der Form $x = \phi(y, z)$ ermitteln die aktuelle Version der Variablen. Der Wert wird ermittelt, indem die ϕ-Funktion x den Wert von y zuweist, wenn der Kontrollfluss aus dem linken Zweig kommt oder x den Wert z zuweist, sofern der Kontrollfluss aus dem rechten Zweig kommt. Die ϕ-Funktion besitzt so viele Argumente, wie Kontrollflüsse die Funktion erreichen. In Abbildung 4.17a ist ein Kontrollflussgraph abgebildet, welcher zwei Kontrollflüsse besitzt.

Das *Conditional* spaltet den Kontrollfluss in zwei einzelne Kontrollflüsse auf, die in Abhängigkeit der Bedingung ausgeführt werden. In der Abbildung 4.17b entstehen so zwei Versionen der Variablen v, etwa v_2 und v_3. Die ϕ-Funktion berechnet die aktuelle Version und weist sie der Variablen v_4 zu. ϕ-Funktionen werden nicht nur



(a) Kontrollflussgraph mit zwei möglichen Kontrollfüssen

(b) Kontrollflussgraph mit ϕ-Funktion

Abbildung 4.17. Die SSA - ϕ-Funktion angewendet auf einen einfachen Kontrollfluss

bei *Conditionals*, sondern auch bei Schleifen benötigt.

Die Berechnung der ϕ-Funktion ist nicht trivial. Rosen, Wegman und Zadeck [RWZ88] geben einen relativ direkten Weg an, die Phi-Funktion einzufügen, während Reif und Tarjen [RT82] einen aufwändigeren Algorithmus mit geringer Zeitkomplexität angeben, welcher auf Bit-Vektor Operationen beruht. Brandis und Mössenbeck stellen einen Single-Pass Generation für SSA [BM94] vor. In dieser Arbeit wird die Idee von Rosen, Wegman und Zadeck [RWZ88] verfolgt, da sie einen Weg vorstellen, der sich mit einigen Anpassungen gut auf SCL Code übertragen lässt.

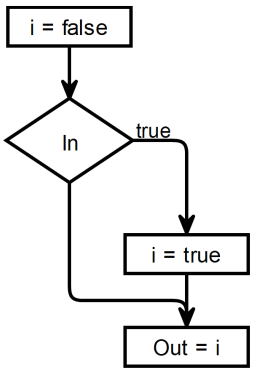
Mit Hilfe der SSA-Transformation können die Datenabhängigkeiten der einzelnen Variablen aufgelöst werden. Da jede Variable nur noch einmal zugewiesen wird, kann der SSA transformierte SCL-Code in Hardware übersetzt werden. Im folgenden Abschnitt werden die Eigenschaften der Transformation erläutert.

4.4.2 SCL zu SSA SCL Transformation

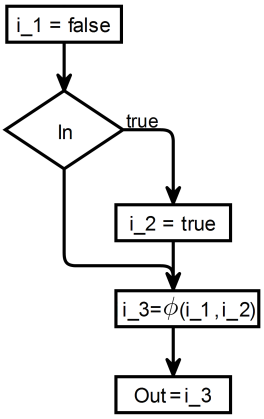
Bei der Transformation eines SCL-Programms kann die Umbenennung der Variablen in eine versionierte Variable sowohl auf der linken als auch der rechten Seite einer Zuweisung direkt erfolgen. Für jede Variable wird die aktuelle Versionsnummer gespeichert. Wird auf eine Variable schreibend zugegriffen, wird eine neue Variable mit der nächst-höheren Versionsnummer angelegt. Bei jedem lesenden Zugriff auf eine Variable wird auf die Variable mit der aktuellen Version zugegriffen.

Im sequentiellen SCL-Code sind weder Schleifen noch goto-Sprünge vorhanden. Die einzige Kontrollflussverzweigung wird von bedingten Sprüngen, *Conditionals* hervorgerufen. Es müssen keine komplexen Schleifenberechnungen für ϕ-Funktionen stattfinden.

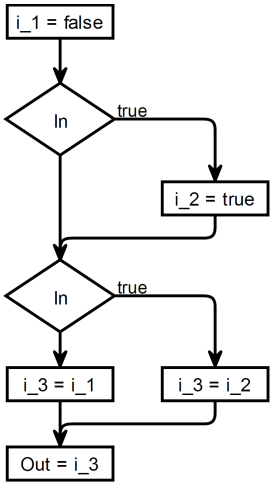
Die Berechnung der ϕ-Funktionen soll anhand der Abbildung 4.18 erläutert werden. Es handelt sich um ein einfaches Programm, in dem es eine Eingabevariable In und eine Ausgabevariable Out, sowie eine lokale Variable i gibt, siehe Abbildung 4.18a. Die Variable Out folgt der Eingabe In mittels einer Hilfsvariablen i. In der zweiten Abbildung 4.18b wurde die SSA-Form, wie oben beschrieben, angewendet. Für die Variable i wurden zwei Versionen i_1 und i_2 angelegt. Nach dem Conditional, welcher den Kontrollfluss aufteilt, wurde die ϕ-Funktion angelegt. Die ϕ-Funktion berechnet an der Stelle den Wert der Variablen i_3. Der Wert ist abhängig davon, welchen Wert In hat, denn In bestimmt den Kontrollfluss. i_3 kann somit den Wert von i_1 oder i_2



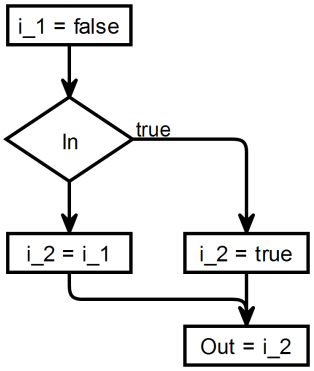
(a) Einfacher Kontrollflussgraph



(b) Kontrollflussgraph mit ϕ-Funktion



(c) Kontrollflussgraph mit expandierter ϕ-Funktion



(d) Kontrollflussgraph mit optimierter ϕ-Funktion

Abbildung 4.18. Die ϕ-Funktion angewendet auf einen einfachen Kontrollflussgraphen mit Umsetzung und Optimierung der ϕ-Funktion

annehmen. Die Zuweisung an die Variable Out ist der Wert der letzten Version von i, hier i_3.

Die ϕ-Funktion kann als eine bedingte Entscheidung umgesetzt werden. Da der Kontrollfluss abhängig von der Bedingung des Conditionals ist, ist die ϕ-Funktion ebenfalls von der gleichen Bedingung anhängig. Sie überprüft mit derselben Bedingung, welche Version der Variablen die aktuellste ist. In Abbildung 4.18c ist die ϕ-Funktion als *Conditional* aufgelöst und in den Kontrollfluss eingefügt worden.

Diese Umsetzung der ϕ-Funktion kann noch optimiert werden. Die beiden aufeinanderfolgenden *Conditional* können zusammengefasst werden, da sie die gleiche Bedingung haben. Abbildung 4.18d zeigt die optimierte Version der ϕ-Funktion. Durch die Optimierung konnte eine weitere Version der Variablen i eingespart werden.

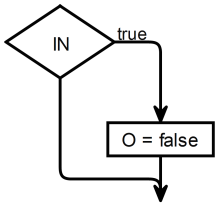
Besonderheiten

Die oben erklärte SSA-Form kann auf Programme angewendet werden, die auf einfachen Strukturen mit lokalen Variablen, Sprüngen und Schleifen beruhen. Bei SCCharts gibt es neben lokalen Variablen jedoch noch Ein- und Ausgabevariablen, sowie Ein-Ausgabevariablen. Diese Variablen müssen gesondert betrachtet werden. Gleiches gilt für Variablen, die den pause-Befehl widerspiegeln.

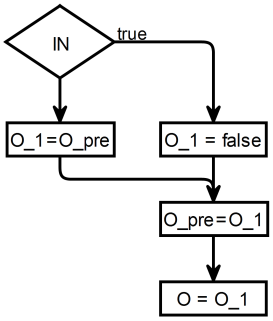
Eingabevariablen SCCharts besitzen Eingabevariablen. Da auf Eingaben nur lesend zugegriffen wird, müssen Eingaben bei der SSA-Transformation nicht transformiert werden. Sie stehen nur auf der rechten Seite in Zuweisungen. Da keine Versionen dieser Variablen benötigt werden, kann bei jedem Zugriff direkt die Eingabevariable selbst gelesen werden.

Ausgabevariablen Ausgaben werden als Reaktionen auf Eingaben vom Programm berechnet und an die Umgebung gegeben. Während der Berechnung dieser Werte können verschiedene Zwischenwerte für Ausgaben entstehen. Aus diesem Grund müssen die Zuweisungen an Ausgabevariablen versioniert werden und am Ende der Tick-Funktion wird die zuletzt berechnete Version der eigentlichen Ausgabevariablen zugewiesen.

In dem Programm in Abbildung 4.19a wird der Ausgang O in Abhängigkeit einer Bedingung In geschrieben. Das bedeutet, dass beim Erzeugen der SSA-Form entschieden werden muss, welchen Wert eine Ausgabevariable erhält, in dem Fall, dass In false ist. Eine Lösung ist es, dem Ausgang im initialen Tick einen Initialwert zuzuweisen. Gab es hingegen schon einen vorhergehenden Tick, indem der Ausgang beschrieben wurde, wird dem Ausgang der Wert zugewiesen werden, den er im vorherigen Tick hatte. Dazu muss der Wert aus dem vorherigen Tick in einer Hilfsvariablen, einer pre-Variablen, gespeichert werden. In Abbildung 4.19b ist gezeigt, wie O_1 im *false*-Zweig der Wert aus dem vorhergehenden Tick zugewiesen wird. Der Wert aus dem vorherigen Tick entspricht der letzten Version der Variablen, die im vorherigen



(a) Bedingte Zuweisung einer Ausgabevariable



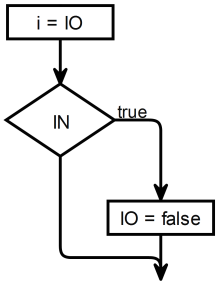
(b) Bedingte Zuweisung zu einer Ausgabevariable in SSA Form

Abbildung 4.19. Die SSA Form angewandt auf ein bedingtes Schreiben einer Ausgabevariable

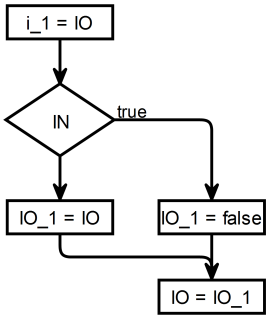
Tick, am Ende der Tick-Funktion, in die pre-Variable gespeichert wurde, siehe O_pre. Der Ausgabevariablen O wird am Ende der Funktion der Wert der höchsten Version zugewiesen. pre-Variablen müssen im initialen Tick initialisiert werden, damit sie auch im ersten Tick einen gültigen Wert führen.

Ein-Ausgabevariablen Ein-Ausgabevariablen haben weitere Eigenschaften. Sie können Werte von der Umgebung lesen und Werte an die Umgebung abgeben. Es muss somit unterschieden werden, für welchen Zweck, ob lesend oder schreibend, die Variable gerade verwendet wird. Wird während des Programmverlaufs lesend auf eine Ein-Ausgabevariable zugegriffen, wird, wie bei normalen Eingaben, direkt der Wert aus der Umgebung gelesen, solange die Variable noch nicht geschrieben wurde. Wurde die Variable schon geschrieben, überdeckt dieser Schreibzugriff den Wert aus der Umgebung und es wird der geschriebene Wert gelesen. Bei schreibenden Zugriffen wird wie bei normalen Ausgaben verfahren und Versionen der einzelnen Schreibvorgänge angelegt. Am Ende der Tick-Funktion wird die höchste Version dem eigentlichen Ausgang zugewiesen.

Ein Unterschied besteht im Vergleich zu einfachen Ausgabevariablen, wenn Ein-Ausgabevariablen zum Beispiel abhängig von einer Bedingung geschrieben werden sollen und die SSA-Transformation angewendet wird. Wenn die Bedingung in Abbildung 4.20 den true-Zweig ausführt, wird eine neue Version der Variablen angelegt und der zu schreibende Wert zugewiesen. Bestimmt die Bedingung hingegen, dass der false-Zweig ausgeführt werden soll, wird der Ausgang nicht beschrieben. Der Wert aus der Umgebung bestimmt den Wert der Ausgabe. Denn wenn eine Ein-Ausgabevariable in einem Tick nicht geschrieben wird, hat die Ausgangsvariable den Wert der Umgebung. Es wird bei Ein-Ausgabevariablen somit keine weitere Variable benötigt, die den Wert der Ausgabe aus dem vorhergehenden Tick speichert.



(a) Bedingte Zuweisung einer Ein-Ausgabevariable



(b) Bedingte Zuweisung einer Ein-Ausgabevariable in SSA Form

Abbildung 4.20. Die SSA Form angewandt auf ein bedingtes Schreiben einer Ein-Ausgabevariable

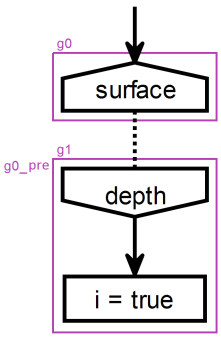
SSA-Form für Pre-Variablen Wie bei der Transformation von SCL zu sequentiell SCL im Abschnitt 4.2.5 bereits dargelegt, wird die pause-Anweisung in die einzelnen Variablen gx (*Surface*) und gx_pre (*Depth*) übersetzt. In dem einfachen SCL-Code aus Abbildung 4.6 sind die erzeugten pause-Variablen zu sehen. Was im Hinblick auf die SSA-Form Generierung auffällt, ist, dass die Variable $g0\_pre$ gelesen wird, obwohl sie vorher noch nicht geschrieben wurde. Es existiert somit noch keine Version dieser Variablen. pre-Variablen halten den Wert aus dem vorhergehenden Tick. In der SSA-Form ist das die letzte Version, die am Ende der Tick-Funktion von der Variablen existiert.

Für pre-Variablen gibt es selbst keine Version, denn sie speichern die letzte Version einer Variablen, damit sie im nächsten Tick zur Verfügung steht. Somit wird bei der Zuweisung der pre-Variable keine neue Version angelegt und die letzte Version der Variablen am Ende der Funktion der pre-Variablen zugewiesen. Der lesende Zugriff auf eine pre-Variable erfolgt immer auf die pre-Variable selbst und nicht auf eine Version von ihr. pre-Variablen müssen im initialen Tick initialisiert werden.

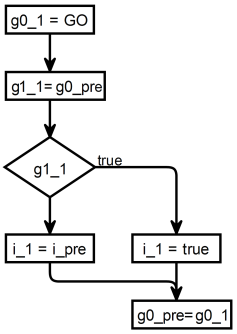
In Abbildung 4.21 ist die SSA-Transformation eines Beispiels gezeigt. Die GO-Variable startet die Ausführung des Programms. Im ersten Tick ist die *Surface* der pause-Anweisung aktiv ($g0\_1$). Dem Signal i wird im zweiten Tick der Wert true zugewiesen, da die *Depth* ($g0\_pre$) und somit das Guard $g1\_1$ aktiv ist. Der Wert der Variablen $g0\_1$ wurde durch die pre-Variable im nächsten Tick zur Verfügung gestellt.

One-Pass-Transformation

Die SSA-Transformation konnte in Form einer *One-Pass-Transformation* implementiert werden. Das bedeutet, um sequentiellen SCL Code in eine SSA-Version zu überführen, muss das SCL-Modell nur einmal durchlaufen werden. Alle einzelnen Transformationsschritte können direkt entschieden und umgesetzt werden. Eine *One-Pass-Transformation* ist möglich, da sequentielle SCL-Programme nur *Conditio-*



(a) Kontrollflussgraph einer Pause Anweisung mit Basic Blocks



(b) Kontrollflussgraph einer Pause Anweisung in SSA Form

Abbildung 4.21. Die SSA Form angewendet auf eine Pause Anweisung unter Verwendung der Basic Blocks

nals als Kontrollsflussverzweigungen besitzen und keine *goto*-Sprünge vorhanden sind [BM94]. Mit *goto*-Sprüngen kann, im Gegensatz zu *Condtitional*-Sprüngen, entgegen der normalen Kontrollflussrichtung gesprungen werden, was die Entwicklung einer One-Pass-Transformation erschwert. Die Komplexität ist abhängig von der Anzahl der Anweisungen, den Knoten im Kontrollflussgraphen, die ein Programm hat. Die Komplexität beläuft sich damit auf $O(n)$. In Listing 4.4 ist das vollständig transformierte Beispiel aus Abbildung 4.2 in der SSA-Form zu sehen.

4.4.3 SSA SCL zu VHDL Transformation

In dem zweiten Teil der Transformation wird die Transformation von SSA SCL Programmen zu einer VHDL Entity erläutert. Zu Beginn des Abschnitts werden grundlegende Eigenschaften zum VHDL Design vermittelt und im weiteren Verlauf wird auf die Einzelheiten der Transformation eingegangen.

VHDL-Programmierung

In VHDL [Inc] werden sogenannte *Entities* beschrieben, sie bilden die Funktionen von Schaltkreisen ab. Eine Übersicht einer VHDL-Entity ist in Abbildung 4.22 gezeigt. Entities können weitere Entities enthalten, die *Components* (Komponenten) genannt werden. Die oberste Entity wird als *Top-Level-Entity* bezeichnet. Jede Entity besitzt eine Entitiy *Declaration* (Deklaration) und eine *Architecture* Beschreibung, im Folgenden Architekturbeschreibung genannt. Die Entity Deklaration beschreibt das Interface der Entity mit seinen Ein- und Ausgaben, um mit der Außenwelt zu kommunizieren. Die Beschreibung des Verhaltens einer Entity ist eine Zusammenstellung aus verbundenen Komponenten, Prozessen und Befehlen und befindet sich in der Architekturbeschreibung.

```
1  module ABO_tick_ssa
2  input output boolean A;
3  input output boolean B;
4  output boolean O1;
5  output boolean O2;
6  // local definitions
7  {
8     g0_1 = RESET;
9     if g0_1 then
10       O1_1 = false;
11       O2_1 = false;
12       else O1_1 = O1_pre;
13       O2_1 = O2_pre;
14    end;
15    g5_1 = g4_pre;
16    g7_1 = g6_pre;
17    g2_1 = g0_1 || g5_1;
18    g3_1 = g2_1 && A;
19    if g3_1 then
20       B_1 = true;
21       O1_2 = true;
22       else B_1 = B;
23       O1_2 = O1_1;
24    end;
25    g4_1 = g2_1 && ! A;
26    g8_1 = g7_1;
27    g9_1 = g8_1 && B_1;
28    if g9_1 then
29       O1_3 = true;
30       else O1_3 = O1_2;
31    end;
32    g6_1 = g0_1 || ( g8_1 && ! B_1 );
33    e2_1 = ! ( g4_1 );
34    e6_1 = ! ( g6_1 );
35    g1_1 = ( g3_1 || e2_1 ) && ( g9_1 || e6_1 ) && ( g3_1 || g9_1 );
36    if g1_1 then
37       O1_4 = false;
38       O2_2 = true;
39       else O1_4 = O1_3;
40       O2_2 = O2_1;
41    end;
42    g4_pre = g4_1;
43    g6_pre = g6_1;
44    O1_pre = O1_4;
45    O2_pre = O2_2;
46    A = A;
47    B = B_1;
48    O1 = O1_4;
49    O2 = O2_2;
50 }
```

Listing 4.4. ABO in sequentiellen SSA SCL Code

Entity Deklaration Bei der Entity Deklaration handelt es sich um die Interfacebeschreibung der Entity. Sie beschreibt die vorhandenen Ein- und Ausgänge, die zur Kommunikation mit weiteren Komponenten oder der Umgebung dienen. Dabei können Eingangs- und Ausgangssignale in Form von einzelnen Signalen oder

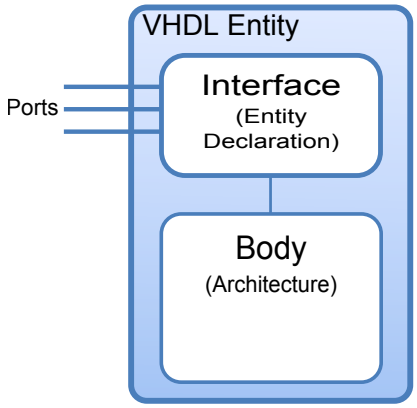


Abbildung 4.22. Struktur einer VHDL Entity

Signalbussen angelegt werden. Jedes Signal besitzt einen eindeutigen Typ. Die Beschreibung des Interfaces wird dabei aus der Deklaration der SCL-Modells gewonnen, siehe Abbildung 4.13, und in einer *Port*-Beschreibung definiert. Die Syntax der Entity und *Port*-Deklaration ist nachfolgend aufgeführt.

$$
\begin{aligned}
&entity\ NAME\_OF\_ENTITY\ is\ [generic\ generic\_declarations);]\\
&\qquad port(signal\_names:\ mode\ type;\\
&\qquad signal\_names:\ mode\ type;\\
&\qquad\quad :\\
&\qquad signal\_names:\ mode\ type);\\
&end\ [NAME\_OF\_ENTITY];
\end{aligned}
$$

Eine Entity besitzt einen Namen und die Deklaration wird mit dem Schlüsselwort port eingeleitet. Darin werden die Signale mit Namen, Typ (*type*) und Richtung (*mode*) (Ein- oder Ausgang) deklariert. Als Typen können zum Beispiel *boolean*, *integer* und *std_logic* verwendet werden. *std_logic* ist ein Datentyp mit dem Werte auf Leitungen abgebildet werden und kennt neun Zustände, darunter sind unter anderem logisch High und Low, Hochohmig, Kurzschluss und Don't Care. Dieser Typ wird überwiegend für die Simulation verwendet, um die verschiedenen Signalzustände sichtbar zu machen.

VHDL-Architektur Die Architekturbeschreibung definiert wie sich der Schaltkreis verhalten soll und wie er implementiert ist. Die Beschreibung kann dabei aus einer Verhaltensbeschreibung bestehen, in der die Funktion direkt beschrieben wird oder aus einer Strukturbeschreibung, die beschreibt, wie Komponenten untereinander verbunden sind. Eine Mischung beider Beschreibungsformen ist ebenfalls möglich.

Die programmierten Befehle oder Komponenten werden entgegen der sequentiellen Programmierreihenfolge parallel ausgeführt. In einer Architekturbeschreibung können unter anderem intern verwendete Signale, Konstanten, Komponenten und Funktionen definiert werden. Nach dem Schlüsselwort begin beginnt die Beschreibung des Schaltkreises. Nachfolgend ist die Syntax der Architekturbeschreibung aufgeführt.

$$
\begin{aligned}
&architecture\ architecture\_name of NAME\_OF\_ENTITY\ is\\
&\quad --\ Declarations\\
&\quad\quad :\\
&\quad begin\\
&\quad --\ Statements\\
&\quad\quad :\\
&end\ architecture\_name;
\end{aligned}
$$

Innerhalb der Architekturbeschreibung können nur Signale verwendet werden. Signale bilden die Funktion von Leitungen ab. Werte, die Signalen zugewiesen werden, sind erst nach einer Verzögerung gültig. Die Verzögerung kann bei der Signalzuweisung angegeben werden, wird kein Wert angegeben, ist der Wert nach einer δ Verzögerung gültig.

Das VHDL Prozess Konstrukt Das VHDL-Prozess-Konstrukt ist für die sequentielle Programmierung von Schaltkreisen ausgelegt. Es beschreibt das Verhalten eines Schaltkreises über die Zeit. Einem Prozess liegt die folgenden Syntax zugrunde:

$$
\begin{aligned}
&[process\_label :]\ process\ [(sensitivity\_list)]\ [is]\\
&\quad [process\_declarations]\\
&\quad\quad begin\\
&\quad\quad\quad list\ of\ sequential\ statements\ such\ as :\\
&\quad\quad\quad\quad signal\ assignments\\
&\quad\quad\quad\quad variable\ assignments\\
&\quad\quad\quad\quad case\ statement\\
&\quad\quad\quad\quad exit\ statement\\
&\quad\quad\quad\quad if\ statement\\
&\quad\quad\quad\quad loop\ statement\\
&\quad\quad\quad\quad next\ statement
\end{aligned}
$$

$$
\begin{array}{c}
\textit{null statement} \\
\textit{procedure call} \\
\textit{wait statement} \\
\textit{end process } [\textit{process\_label}];
\end{array}
$$

Innerhalb von Prozessen können diverse Programmierkonstrukte verwendet werden. Diese sind speziell für die sequentielle Programmierung ausgelegt. Es stehen zum Beispiel Konstrukte wie if-Anweisungen, Schleifen und *switch-case*-Anweisungen zur Verfügung, wie sie aus C oder Java bekannt sind. Im Gegensatz dazu, dass in der Regel Signale zur Programmierung verwendet werden, können in Prozessen auch Variablen benutzt werden.

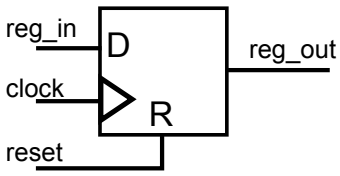
Ein Prozess wird innerhalb der Architektur, parallel zu anderen Anweisungen, deklariert. Innerhalb des Prozesses ist die Ausführung dennoch sequentiell. Werte können innerhalb des Prozesses an extern deklarierte Signale zugewiesen werden und somit mit anderen Komponenten der Architektur kommunizieren. In der Sensitivitätsliste werden Signale aufgeführt, auf die der Prozess reagieren soll. Jede Änderung eines sensitiven Signals veranlasst den Prozess, sofort die Ausführung des internen Codes zu beginnen. Die Sensitivitätsliste ist optional. Wenn die Liste nicht angegeben wird, muss eine wait until-Anweisung definiert werden, damit die Prozessausführung gestartet wird und nicht kontinuierlich läuft. In der wait-Anweisung können Signale und Verknüpfungen zwischen Signalen angegeben werden. Die Sensitivitäsliste und die wait-Anweisung können nicht gleichzeitig verwendet werden. Deklarationen von Konstanten und Variablen müssen vor dem Schlüsselwort begin in den Prozess-Deklarationen erfolgen.

Die Zuweisung von Werten an Variablen geschieht sofort ($\delta = 0$) und wird mit dem := Operator gekennzeichnet. Das bedeutet, dass sequentielle Zuweisungen wie im Beispiel 4.3 innerhalb eines Prozesses möglich sind. Bei Signalen in Prozessen hingegen werden die Zuweisungen erst am Ende eines Prozesses übernommen. Somit wird der Wert auf das Signal geschrieben, welches als letztes zugewiesen wurde. Bei einer Zuweisung mit anschließender Überprüfung des Wertes muss daher beachtet werden, dass im aktuellen Durchlauf der Wert aus dem letzten Durchgang getestet wird.

Register Register werden verwendet, um Werte innerhalb der Hardware zu speichern. Werte auf Leitungen stabilisieren sich in der Regel nach einer Zeit t, wenn keine Änderungen anderer Signale diesen Wert beeinflussen. Der Wert ist jedoch nicht gespeichert. Wenn Werte explizit gespeichert werden sollen, werden Register benötigt. Register werden mit Hilfe von Prozessen erzeugt. Die Abbildung 4.23 zeigt ein Register. Der Prozess wird bei jeder steigenden Flanken des clock Signals gestartet und speichert den Wert am Eingang D, das wird durch wait until rising_edge(clock)

```
1 register: process
2  begin
3  wait until rising_edge(clock);
4     if(reset = true) then
5          reg_out <= false;
6     else
7          reg_out <= reg_in;
8     end if;
9 end process;
```

(a) VHDL Code zum Erzeugen eines Registers mit taktsynchronen Reset



(b) Register als Schaltsymbol

Abbildung 4.23. Ein einfaches Register beschrieben in VHDL und dessen grafische Repräsentation

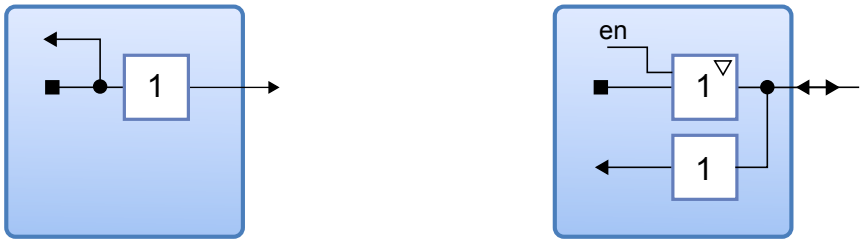
erreicht. Wenn das Resetsignal reset vorhanden ist, wird das Register zurückgesetzt. Ist das Resetsignal nicht vorhanden, übernimmt das Register bei jeder steigenden Taktflanke den Wert am Eingang D.

Transformationsdetails

Die Transformation von SSA SCL Programmen erfolgt nicht, wie in der ersten Transformation, Unterkapitel 4.3, nur mit Hilfe des Prozess Konstruktes. Bei diesem Ansatz werden verschiedene SCL-Codeabschnitte einzeln transformiert. So werden Teile des Codes direkt in Hardware transformiert und das Prozess-Konstrukt nur verwendet, um Register zu erzeugen. In diesem Kapitel wird der SSA SCL Code von ABO aus Listing 4.4 übersetzt. Weiterhin werden Besonderheiten wie Sampling-Register und die *GO*-Signal Generierung erklärt.

Transformation von Ein-Ausgabevariablen In Unterkapitel 4.4.3 wurde die Entity Deklaration vorgestellt, in der die Ein- und Ausgangssignale einer Entity festgelegt werden. Bei einem SCChart können in der Interface Deklaration Eingabe-, Ausgabe- und Ein-Ausgabe-Variablen definiert werden. Eingaben können nur Werte aus der Umgebung lesen und Ausgaben nur Werte an die Umgebung abgeben bzw. schreiben. Ein-Ausgabevariablen hingegen können Werte von der Umgebung lesen und Werte an die Umgebung schreiben. In VHDL können in der Port Deklaration Ein- und Ausgaben (*in* bzw. *out*) sowie Buffer (*buffer*), Abbildung 4.24a oder Ein-Ausgaben (*inout*), Abbildung 4.24b definiert werden. Ein Buffer ist ein Ausgabesignal welches auch innerhalb der Entity gelesen werden darf. Ein VHDL-inout-Signal kann gelesen und geschrieben werden.

Die Ein- und Ausgabevariablen können direkt in VHDL Ein- und Ausgangssignale übersetzt werden. Ein-Ausgabevariablen lassen sich nicht direkt nach VHDL übersetzen. Ein Buffer-Signal kann nicht verwendet werden, da es kein Signal aus der



(a) Portdeklaration *buffer* — Grafische Darstellung

(b) Portdeklaration *inout* — Grafische Darstellung

Abbildung 4.24. Besondere Portdeklarationen in VHDL

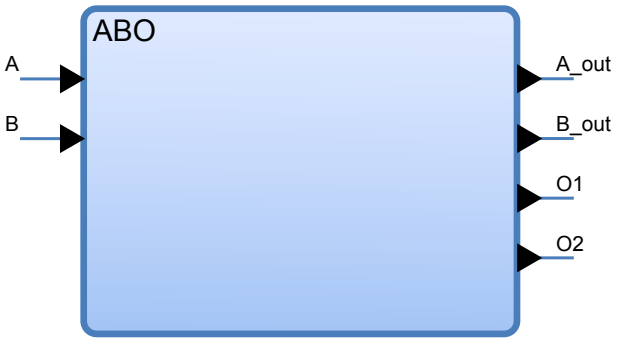
Umgebung lesen kann, sondern nur das intern geschriebene Signal. Die Verwendung eines VHDL-inout-Signals ist ebenfalls nur schwer möglich, denn es müssen Kontrollmechanismen entworfen werden, die sicherstellen, dass entweder die Umgebung ein Signal auf die Leitung schreibt oder das System selbst. So kann die Umgebung für ein Signal den Wert false vorsehen, das System jedoch auf dieselbe Leitung den Wert true schreiben. Es würde ein Kurzschluss entstehen.

Eine Alternative, mit Ein-Ausgabevariablen der SCCharts zu verfahren, ist, sie direkt in getrennte Ein- und Ausgangsleitungen zu übersetzen. Dabei behält das Eingangssignal den Namen der Ein-Ausgabevariablen und das Ausgangssignal erhält den gleichen Namen mit der Erweiterung $\_out$. In Abbildung 4.25 ist die Transformation der Entity Deklaration mit Hilfe des SSA SCL Codes und dem entsprechenden VHDL-Codes sowie einer Übersicht gezeigt.

Speichern von Eingabesignalen Die Eingangssignale eines SCCharts werden direkt mit der internen Logik, die aus der Beschreibung des SCCharts hervorgeht, verbunden. Sie beeinflussen somit unmittelbar die Berechnung der Ausgaben. Damit sichergestellt ist, dass in jedem Tick das richtige Ergebnis berechnet wird, müssen die Eingangssignale während eines Ticks stabil sein und dürfen sich nicht ändern. In Abbildung 1.2 werden die Eingaben zu Beginn eines Ticks gelesen. Auf Grundlage dieser gelesenen Werte findet die Berechnung statt. Im Gegensatz zur Softwareausführung werden keine Variablen geschrieben und die Ausgaben berechnet, sondern die Logik berechnet kontinuierlich auf Basis der Eingaben die Ausgaben. Damit die Werte während der Berechnung konstant bleiben, werden sie zu Beginn des Ticks in Registern gespeichert, den sogenannten Sampling-Registern.

In Abbildung 4.26 sind für die Ein-Ausgabevariablen A und B von ABO Sampling-Register erstellt worden.

Transformation für Pre-Werte Sollen Signalwerte zwischen Ticks gespeichert werden, müssen Register angelegt und verwendet werden. Im Gegensatz zur naiven Lösung müssen benötigte Register explizit angelegt werden, denn es werden in



(a) ABO Übersicht — Port Deklaration

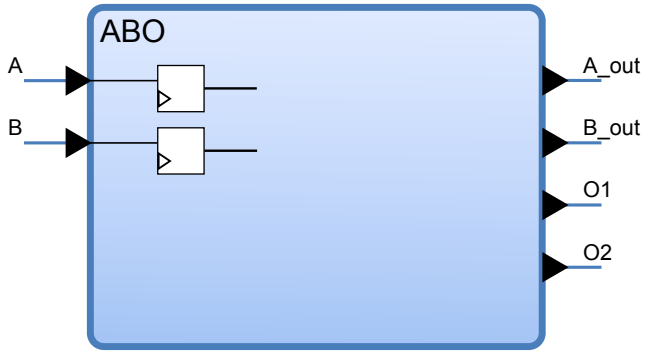
```
1  module ABO_tick_ssa
2  input output boolean A;
3  input output boolean B;
4  output boolean O1;
5  output boolean O2;
6  // local definitions
```

(b) ABO SSA SCL Code — Port Deklaration

```
1  ENTITY ABO IS
2  PORT(
3       -- inputs
4       A: IN boolean;
5       B: IN boolean;
6       -- outputs
7       A_out : OUT boolean;
8       B_out : OUT boolean;
9       O1 : OUT boolean;
10      O2 : OUT boolean
11     );
12 END ABO;
```

(c) ABO VHDL Code — Port Deklaration

Abbildung 4.25. ABO - Interface Deklaration



(a) ABO Übersicht — Sampling Register

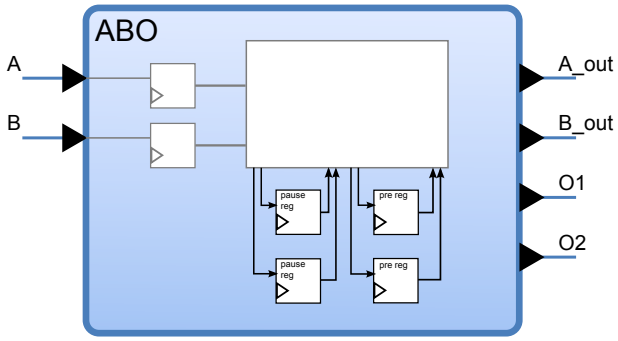
```
65 inputRegister: process
66 begin
67 wait until rising_edge(tick);
68     A_local <= A;
69     B_local <= B;
70 end process;
```

(b) ABO VHDL Code — Generierung der Sampling Register

Abbildung 4.26. ABO - Generierung der Sampling Register

diesem Lösungsansatz Signale anstelle von Variablen verwendet. Der Compiler kann nicht entscheiden, welche Signale in Registern gespeichert werden müssen.

Im SCL-Code ist durch Variablenbezeichnung $\_pre$ festgelegt, welche Werte über Tickgrenzen hinweg erhalten bleiben müssen. Es muss somit für jede pre-Variable ein Register angelegt werden. Der Wert, der gespeichert wird, ergibt sich aus der Zuweisung am unteren Ende des sequentiellen SCL-Codes.



(a) ABO Übersicht — mit Pre-Registern

```
42 g4_pre = g4_1;
43 g6_pre = g6_1;
44 O1_pre = O1_4;
45 O2_pre = O2_2;
```

(b) ABO SSA SCL Code — Zuweisung von Pre Werten

```
49 registers: process
50 begin
51 wait until rising_edge(tick);
52    g4_pre <= g4_1;
53    g6_pre <= g6_1;
54    O1_pre <= O1_4;
55    O2_pre <= O2_2;
56 end process;
```

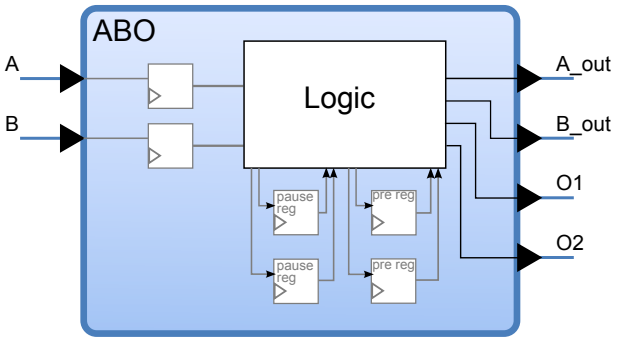
(c) ABO VHDL Code — Register für Pre Werte

Abbildung 4.27. ABO — Generierung von Pre Registern

Ein Register wird mit Hilfe eines Prozesses, wie in Abschnitt 4.4.3 beschrieben, angelegt. Die pre-Register Generierung ist in Abbildung 4.27 zu sehen. Dazu sind die SSA SCL Befehle aufgeführt, die in Register transformiert werden sollen. Ebenfalls ist der passenden VHDL Code abgebildet. In der Übersicht sind die entstandenen Register ebenfalls eingezeichnet.

Logik Transformation Der noch nicht transformierte SCL-Code kann nun fast vollständig in VHDL übersetzt werden, eine Ausnahme stellen die Zuweisungen an die Ausgaben dar. Zum übrigen Code gehört die eigentliche Beschreibung des Verhaltens. Dazu werden für alle verwendeten Variablen im sequentiellen SCL-Code Signale angelegt. Der SCL-Code kann unter Berücksichtigung der VHDL-Syntax transformiert werden. Zuweisungen für Signale werden mit dem Operator $<=$ und logische Operationen wie $\&\&$, $||$ und ! werden in VHDL mit and, or und not implementiert. Da die if-Anweisung für sequentielle Berechnungen reserviert ist, werden bedingte Anweisungen in eine $when - else$-Anweisung übersetzt, ein Beispiel ist in Abbildung 4.38 zu sehen.

Zuweisung von Ausgabesignalen Bei der Zuweisung der berechneten Werte an die Ausgabeleitungen muss beachtet werden, um welche Art Variable es sich in dem SCChart gehandelt hat. Normalen Ausgabesignalen wird die höchste SSA-Version der Variablen zugewiesen, siehe O1 und O2 in Abbildung 4.28. Da Ein-Ausgabevariablen in zwei separate Leitungen aufgetrennt wurden, müssen die Ausgaben an das Signal mit der jeweiligen $\_out$ Erweiterung gegeben werden, siehe A_out und B_out. A_out wird der Wert aus dem Sampling-Register A_local zugewiesen, da A intern nicht



(a) ABO Übersicht — mit Logik und Zuweisung der Ausgaben

```
46 // logic implementation hidden
47
48 A = A;
49 B = B_1;
50 O1 = O1_4;
51 O2 = O2_2;
```

(b) ABO SSA SCL Code — Zuweisung der Ausgaben

```
42 -- logic implementation hidden
43
44 A_out <= A_local;
45 B_out <= B_1;
46 O1 <= O1_4;
47 O2 <= O2_2;
```

(c) ABO VHDL Code — Zuweisung der Ausgaben

Abbildung 4.28. ABO - Zuweisen der Ausgaben

geschrieben wird. B_out bekommt die höchste Version des Wertes zugewiesen, da B intern geschrieben wird.

Das Tick Signal Die Berechnungen in einem SCChart werden in jedem Tick angestoßen. Es werden die Eingaben gelesen und die Ausgaben, ohne Zeit zu verbrauchen, berechnet, siehe Unterkapitel 1.1. Das funktioniert in der Form nur im Modell. Die Hardware benötigt eine gewisse Zeit, die Eingaben einzulesen und die Ausgaben zu berechnen.

Die minimale Zeit, die ein Schaltkreis zum Berechnen der Ausgaben benötigt, ist abhängig von den Gatter-Laufzeiten der Bausteine und vom längsten Pfad der Schaltung. Die Gatter Laufzeiten sind wiederum abhängig vom Synthesevorgang und vom *Place-and-Route*-Algorithmus des Compilers. Der *Place-and-Route*-Algorithmus berechnet wie der synthetisierte Schaltkreis auf den jeweiligen FPGA programmiert werden kann. In dieser Arbeit wurde nicht das Ziel verfolgt, eine minimale Tick-Dauer zu ermitteln. Deshalb wurde Tick in Form eines periodischen Taktsignals realisiert, dessen Periodendauer lang genug ist, um alle Berechnungen der in dieser Arbeit verwendeten Modelle durchzuführen. Die Periodendauer wurde auf 100ns festgelegt. Dazu wird in jeder Entity ein weiteres Eingangssignal $tick$ generiert, siehe Abbildung 4.29.

Das tick-Signal wird benötigt, um die Berechnungen innerhalb der Schaltung zu starten. Dabei wird das Signal verwendet, um alle internen Register zu takten. Zu den Registern zählen Sampling- und pre-Register.

Mit jeder steigenden Flanke des tick-Signals werden so die Eingangssignale in den Sampling-Registern gespeichert und die pre-Register übernehmen den Wert aus dem

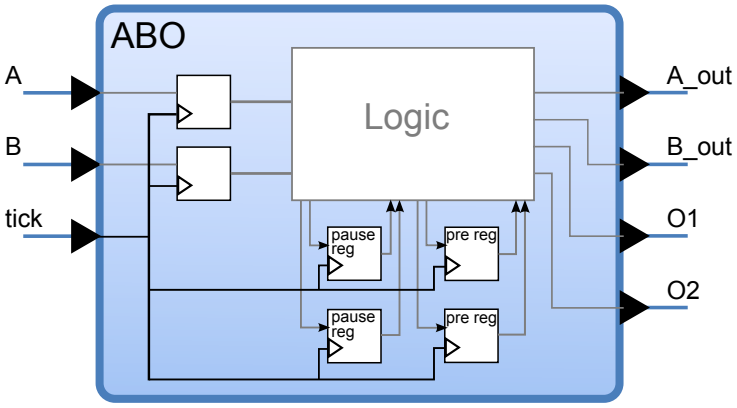


Abbildung 4.29. ABO Übersicht — mit Tick-Signal

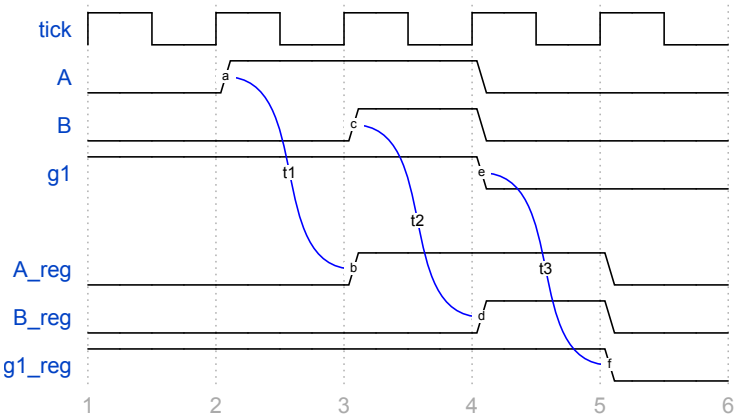


Abbildung 4.30. Tick Signal Diagramm — Wann werden Werte in Registern gespeichert

vorhergehenden Tick. In Abbildung 4.30 ist ein Signaldiagramm gezeigt, welches die Funktion des tick-Signals verdeutlicht. Als Eingaben sind A und B dargestellt. g1 ist ein lokales Signal, dessen Wert in einem Register gespeichert werden soll. Die Bezeichnungen t1, t2 und t3 zeigen die Übernahme der Werte in die Register. A_reg, B_reg und g1_reg zeigen die gespeicherten Werte an. Die Buchstaben (a-f) dienen zur Flankenbeschriftung.

Das Rücksetzen einer Schaltung Ein Reset-Signal wird benötigt, um den synthetisierten Schaltkreis in einen definierten Zustand zu bringen. Dazu werden die verwendeten Register über ein zusätzliches reset-Signal, siehe Abbildung 4.31, zurückgesetzt. Nach dem Rücksetzen der Schaltung soll sie automatisch mit der Ausführung beginnen, wofür ein GO-Signal generiert wird, siehe Abschnitt 4.4.3. Es gibt verschiedene Möglichkeiten, einen Schaltkreis zurückzusetzen und anschließend zu starten. Zwei verschiedene Möglichkeiten werden im nächsten Abschnitt 4.4.4 erklärt.

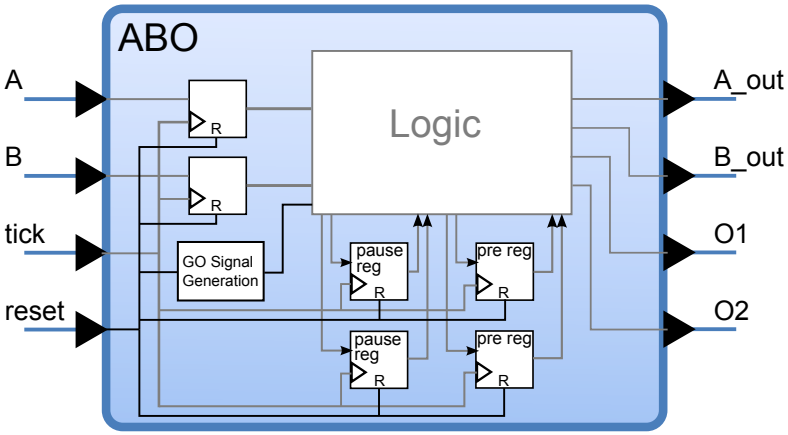


Abbildung 4.31. ABO Übersicht — mit Reset und GO Signal

Das GO-Signal Das tick-Signal veranlasst die Register, die aktuellen Werte zu speichern und stößt somit eine neue Berechnung der Reaktion, für einen Tick an. Diese Berechnung basiert auf dem aktuellen Kontrollfluss, der sich in Registerzuständen widerspiegelt. Da Tick-Signal kann nicht dazu verwendet werden, einen synthetisierten Schaltkreis initial zu starten, d.h. den Kontrollfluss, der in einem SSA SCL Programm existiert, zu beginnen. Das liegt daran, dass das hier verwendete periodische tick-Signal keine weiteren Information zum Starten der Schaltung besitzt. Dazu wird ein weiteres Signal benötigt. Das GO-Signal startet die Ausführung des Kontrollflusses. Nachdem der Schaltkreis zurückgesetzt worden ist und alle Register mit einem initialen Wert geladen sind, soll die Schaltung mit der Ausführung beginnen. Das GO-Signal wird aus dem reset-Signal gewonnen, siehe Abbildung 4.31.

4.4.4 Das Starten und Rücksetzen von transformierten SCCharts

Der Zustand, in dem sich ein synthetisiertes SCChart befindet, wird durch die Werte in den Registern definiert. Ein Zustand wird durch den aktuellen Tick und, sofern vorhanden, durch die auftretenden Eingangssignale bestimmt. Um den Schaltkreis zu Beginn oder während der Laufzeit in einen definierten Zustand zu bringen, müssen diese Register initialisiert werden. Ein externes Reset-Signal kann dazu verwendet werden, einen definierten Wert in alle Register zu laden. Im Folgenden werden zwei Reset-Varianten vorgestellt, die für diesen Einsatz denkbar sind. In der ersten Variante handelt es sich um einen Reset der synchron zum initialen Tick ausgeführt wird. In der zweiten Variante hingegen wird der Reset in einem separaten Tick vor dem initialen Tick ausgeführt. Beide Versionen beinhalten auch eine Generierung des GO-Signals, auf das ebenfalls eingegangen wird.

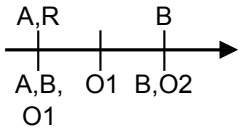


Abbildung 4.32. ABO Ausführungspfad — Erste Reset-Variante

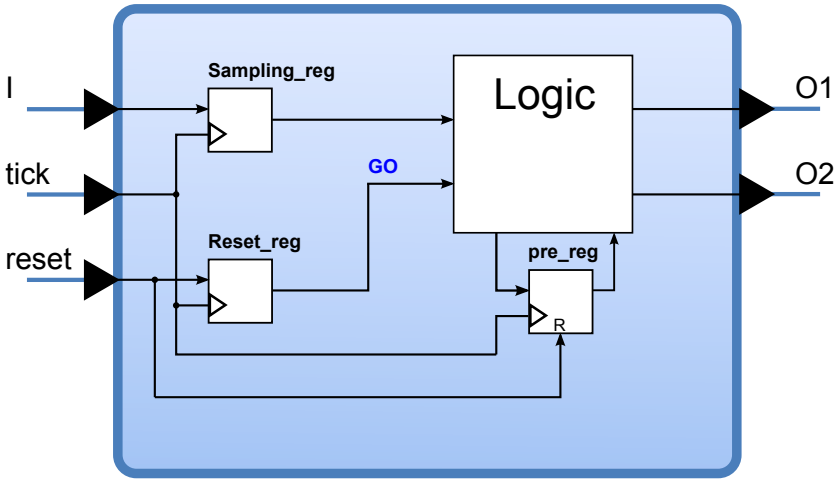


Abbildung 4.33. Schaltplan der ersten Reset-Variante

Erste Reset-Variante

Ziel dieser Reset-Variante ist es, das Rücksetzen und den Start der Berechnung in einem Tick auszuführen. Um diese Variante umsetzen zu können, müssen einige Besonderheiten der verschiedenen Register berücksichtigt werden. Unterschieden werden müssen Sampling-Register und *Pre*-Register. Die Abbildung 4.32 zeigt einen Ausführungspfad von *ABO*, in dem im ersten Tick ebenfalls ein Reset R gesetzt wird.

Der Reset R wird synchron zum initialen Tick gesetzt und ausgeführt. Die internen Register werden zurückgesetzt und die Ausgaben des ersten Ticks berechnet. In Abbildung 4.33 ist ein prinzipielles Schaltbild gezeigt, welches die Eigenschaften dieser Reset-Variante umsetzt. Die Register hinter den Eingängen (A,B,R) sind die Sampling-Register und die übrigen Register sind *Pre*-Register. Das Schaltbild soll die Idee des Resets verdeutlichen, aus dem Grund ist die Logik innerhalb der Schaltung verborgen. Nachfolgend werden die Einzelheiten dieser Variante mit Hilfe des Schaltbildes erklärt.

Sampling Register Sampling-Register verfügen über keinen Eingang zum Rücksetzen des Registers. Sampling-Register dürfen im initialen Tick, in dem auch der Reset ausgeführt wird, nicht zurückgesetzt werden, sondern müssen den Wert aus der Umgebung übernehmen. Würden die Register zurückgesetzt werden, würden die Werte der Umgebung im initialen Tick nicht berücksichtigt werden. Die Berechnung

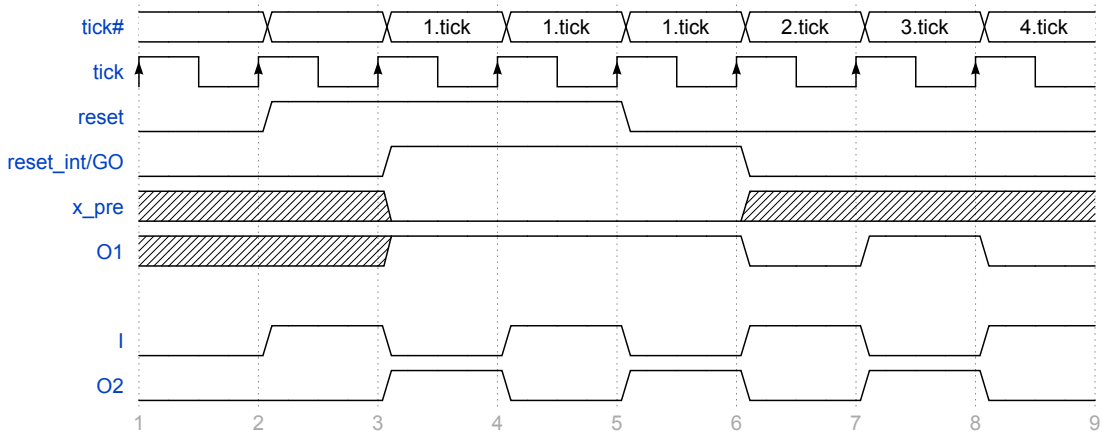


Abbildung 4.34. Signalverlauf der ersten Reset-Variante

würde ein Ergebnis liefern, das in der Regel nicht dem Erwarteten entspricht. Das Reset-Sampling-Register (Reset_reg) besitzt ebenfalls keinen Reset-Eingang, damit der Wert gespeichert werden kann, um daraus später das GO-Signal zu generieren. Eine zweite Variante das Reset-Register umzusetzen, ist, ein Register zu erzeugen, dessen Ein- und Ausgang negiert ist. Am Eingang des Register muss eine Konstante 0 anliegen [Ber02]. Zum Rücksetzten des Registers wird ein taktsynchroner Reset-Eingang benötigt.

***Pre*-Register** Die *Pre*-Register hingegen besitzen einen Reset-Eingang. Sie müssen explizit zurückgesetzt werden, um den Schaltkreis in einen definierten Zustand zu versetzten. Das Rücksetzen dieser Register parallel zum ersten Tick ist problemlos, denn sie speichern Werte aus dem vorhergehenden Tick. Da es sich beim initialen Tick um den ersten Tick handelt, in dem ebenfalls der Reset ausgeführt wird, gibt es keine Werte aus einem vorhergehenden Tick, die gespeichert werden müssen.

GO-Signal Generierung Das GO-Signal wird direkt aus dem Reset-Signal erzeugt. Es wird das gespeicherte Reset-Signal aus dem Reset-Sampling-Register verwendet, um die Berechnung des Schaltkreises zu starten.

Signalverlauf In Abbildung 4.34 ist ein Signalverlauf dieser Reset-Variante aufgeführt. Das Reset-Signal (reset) liegt hierbei über mehrere Ticks an. Das angelegte reset-Signal wird im ersten Tick gespeichert und setzt parallel alle *Pre*-Register (x_*pre*) zurück. Ebenfalls wird das *GO*-Signal synchron zum gespeicherten Reset-Signal erzeugt und startet die Berechnungen. Mit jedem Tick, in dem der Reset aktiv ist, wird der erste Tick erneut berechnet. O1 alterniert und wird im ersten Tick auf true gesetzt. Ab dem zweiten Tick werden die *Pre*-Register entsprechend der Werte aus der Schaltung gesetzt.

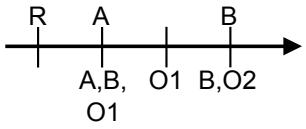


Abbildung 4.35. ABO Ausführungspfad — Zweite Reset-Variante

Eine Besonderheit der ersten Reset-Variante Untersuchungen dieser Variante haben eine Besonderheit dieser Designentscheidung aufgezeigt. In dieser Variante beginnt der Schaltkreis mit der Berechnung im selben Tick wie die Ausführung des Resets. Wenn das Reset-Signal über einen längeren Zeitraum anliegt, berechnet die Schaltung über die Dauer des Resets in jedem Tick erneut den ersten Tick. In der Abbildung 4.34 folgt der Ausgang O2 dem Eingang I. In diesem Beispiel ist I ein alternierendes Signal. Dadurch, dass der erste Tick erneut berechnet wird, wechselt der Ausgang O2 in jedem Tick den Pegel. So führt der Ausgang in jedem ersten Tick einen anderen Wert. Das entspricht zwar dem Verhalten der Schaltung, jedoch ist das Verhalten im Hardwaredesign ungewöhnlich, dass während eines Resets Ausgänge verschiedene Werte führen. Eine verbesserte Lösung wäre, dass alle Ausgänge während der Dauer des Resets einen stabilen Wert durch eine erweiterte Beschaltung der Ausgänge führen. Es könnte zum Beispiel ein Multiplexer am Ausgang erzeugt werden, der während des Resets ein false auf den Ausgang schreibt.

Zweite Reset-Variante

In der ersten Variante wurden der Reset und der initiale Tick synchron zueinander ausgeführt. Die zweite Variante zeigt eine weitere Möglichkeit auf, indem ein zusätzlicher Tick für den Reset verwendet wird, im Folgenden *Reset-Tick* genannt. Diese Variante hat andere Eigenschaften als die erste Variante. In Abbildung 4.35 ist ein Ausführungspfad von *ABO* gezeigt, welcher das Verhalten der zweiten Reset-Variante zeigt. Es ist zu erkennen, dass der Tick für den Reset und der initiale Tick in zwei aufeinander folgende Ticks aufgeteilt wurde. Die Berechnung der Ausgaben geschieht im initialen Tick, nachdem der Reset-Tick aktiv war.

In Abbildung 4.36 ist ein prinzipielles Schaltbild dieser Reset-Variante gezeigt, mit dessen Hilfe die folgenden Eigenschaften dargelegt werden sollen.

Sampling-Register Da für den Reset ein extra Tick vorgesehen ist, müssen im Reset-Tick die Werte aus der Umgebung nicht gespeichert werden. Für ein sauberes Schaltungsdesign werden in diesem Tick alle Sampling-Register zurückgesetzt. Eine Ausnahme stellt das Register dar, welches das Reset Signal speichert. Dieses Register wird nicht zurückgesetzt, wenn ein Reset anliegt, da es diesen Wert speichern soll, um später daraus das GO-Signal zu generieren.

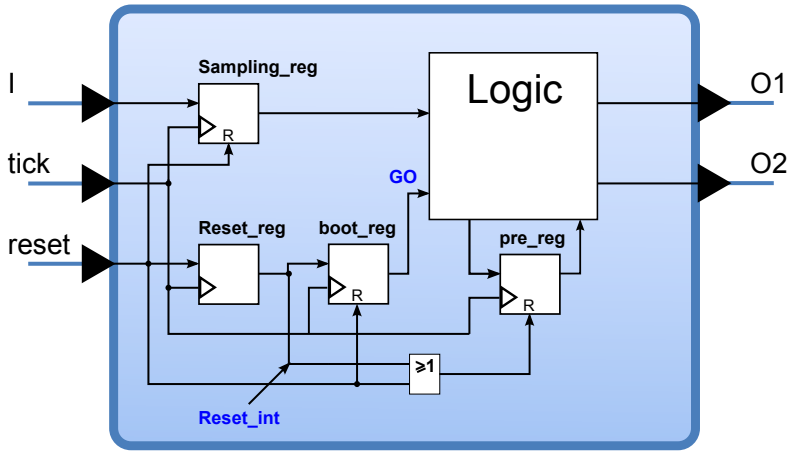


Abbildung 4.36. Schaltplan der zweiten Reset-Variante

***Pre*-Register** An den *Pre*-Registern finden keine Änderungen statt. Sie werden wie in der ersten Variante implementiert. Sie erhalten einen Reset-Eingang, der im Falle des Resets das Register zurücksetzt. Beim Rücksetzen der *Pre*-Register muss darauf geachtet werden, dass sie auch im initialen Tick noch zurückgesetzt sind. Das wird erreicht, indem zum Rücksetzten das gespeicherte Reset-Signal Reset_int verwendet wird.

Sofern das Reset_int Signal zum Rücksetzen verwendet wird, werden die Register im ersten Tick, in dem das reset-Signal anliegt, nicht zurückgesetzt. Es handelt sich um eine Anforderung an das System, das bestimmt, ob die Werte auch in diesem Tick definiert sein sollen. Um ein eindeutiges Verhalten des Systems zu gewährleisten, sollen die *Pre*-Register auch in dem ersten *Reset-Tick* definierte Werte führen. Das kann erreicht werden, indem das reset und das Reset_int Signal über ein *Oder*-Gatter verknüpft die Register zurücksetzt.

GO-Signal Generierung Das GO-Signal wird aus dem Reset-Signal gewonnen, denn der Schaltkreis soll nach dem Reset automatisch die Berechnung starten. Da in dieser Variante das GO-Signal einen Takt später als der Reset benötigt wird, muss das Signal um einen Takt verzögert werden. Signalverzögerungen können mit Registern erreicht werden. In Abbildung 4.36 ist ebenfalls die Generierung des GO-Signals gezeigt.

Das erste Register ist ein Sampling-Register (Reset_reg), welches das Reset-Signal speichert. In jedem Tick wird der Wert des Reset-Signals in das Register übernommen. Das zweite Register (boot_reg) übernimmt im darauffolgenden Takt das Signal vom Reset-Sampling-Register, sofern kein externes Reset-Signal mehr anliegt, und stellt das GO-Signal zur Verfügung. Sofern das externe Reset-Signal noch vorhanden ist, wird des Boot-Register weiterhin zurückgesetzt und kann den Wert am Eingang nicht übernehmen. Es wird kein GO-Signal erzeugt.

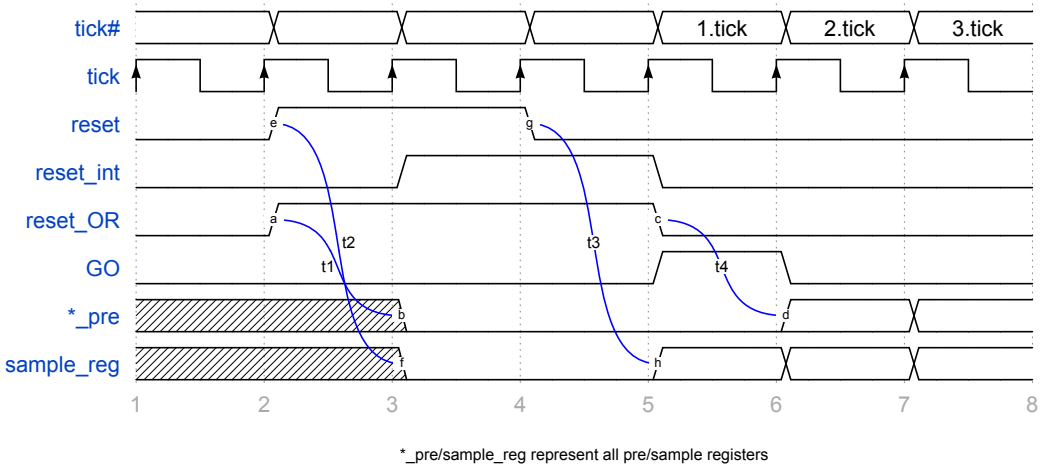


Abbildung 4.37. Signalverlauf der zweiten Reset-Variante mit GO- und Reset-Signal

Reset-Signaldiagramm Der Signalverlauf in Abbildung 4.37 zeigt die einzelnen Signale der zweiten Reset-Variante. Die Erzeugung des GO-Signal ist ebenfalls abgebildet. Diese Schaltung bringt den Vorteil mit sich, dass die Dauer des Resets keinen Einfluss auf das GO-Signal hat. Das GO-Signal ist immer für genau die Dauer eines Ticks nach dem Reset true.

Es muss in dieser Reset-Variante sichergestellt sein, dass im initialen Tick die *Pre*-Register zurückgesetzt sind. Das ist wichtig für die Korrektheit der Berechnung, denn im ersten Tick sind die *Pre*-Werte von Signalen false, weil es keinen *Pre*-Wert gibt.

Informeller Beweis Die Pre-Register sind während des Reset-Ticks und in dem Tick nach dem Reset-Tick mit false initialisiert.
Beweis: Während der Dauer des Resets ist sichergestellt, dass die *Pre*-Register zurückgesetzt werden. Das resultiert aus der Oder-Verknüpfung des reset-Signals und des Reset_int-Signals am Rücksetzeingang eines jeden Pre-Registers. Dieses Signal ist im initialen Tick zur Taktflanke noch vorhanden (Abbildung 4.37 Flanke c). Die Register werden somit im initialen Tick noch einmal zurückgesetzt. Weiterhin berechnet die Schaltung solange keine neuen Werte, wie das GO-Signal nicht vorhanden ist. Das GO-Signal startet den Kontrollfluss der Schaltung. Solange das GO-Signal false ist, wird die Berechnung neuer Werte unterbunden. Während der Taktflanke des initialen Ticks ist das GO-Signal noch nicht vorhanden. Das bedeutet, wenn die *Pre*-Register Werte übernehmen sollen (steigende Taktflanke vorhanden), ist noch kein neuer Wert berechnet worden (GO-Signal noch nicht vorhanden). Zusätzlich können keine neuen Werte übernommen werden, da die Pre-Register in dem initialen Tick noch zurückgesetzt werden, siehe Abbildung 4.37.

Eigenschaft	**erste Reset-Variante**	**zweite Reset-Variante**
Reset im initialen Tick	x	
Ausgänge führen während des Resets definierte Werte		x
Berechnungen währenddessen das Resetsignal anliegt	x	
einmalige GO-Signal Generierung		x
Sample-Register Reset		x
pre-Register Reset	x	x

Tabelle 4.1. Übersicht der Eigenschaften der beiden Reset-Varianten

Vorteile Ein Vorteil ist, dass die Ausgänge während des Resets einen stabilen Wert aufweisen. Da während des Resets die Sampling-Register zurückgesetzt werden, haben die Eingänge während des Resets keinen Einfluss auf die Ausgänge. Das GO-Signal wird erst nach dem Reset-Signal erzeugt, was zur Folge hat, dass während des Resets keine Berechnungen stattfinden. Die Ausgänge führen somit ihren *Pre*-Wert. Dieser Wert wurde durch den Reset zurückgesetzt, wodurch die Ausgänge während der Dauer eines Resets ein false führen.

Fazit

Die erste Reset-Variante hat den Vorteil, dass kein extra Tick für den Reset benötigt wird. Dem gegenüber steht jedoch die Besonderheit, dass die Ausgänge ein instabiles Verhalten aufweisen können, wenn sie zum Beispiel von Eingängen abhängig sind, die während eines Resets ihre Werte ändern. Eine erweiterte Schaltung, die die Ausgänge während des Resets mit einem definierten Wert belegt, kann diese Eigenschaft beheben.

In der zweiten Variante wird ein weiterer Takt benötigt, um das Reset-Signal vom initialen Tick zu trennen. Da das GO-Signal aus dem Reset-Signal generiert wird, wird ein zusätzliches Register zur Verzögerung des Signals benötigt. Der Vorteil dieser Variante ist die klare Trennung zwischen dem Rücksetzen und Starten der Schaltung, sowie die stabilen Ausgänge während des Resets. De Tabelle 4.1 fast alle Eigenschaften der beiden Varianten noch einmal zusammen.

In dieser Arbeit wurde die zweite Variante des Resets implementiert. Die klare Trennung zwischen Reset und initialen Tick sowie die genannten Vorteile haben zu dieser Entscheidung geführt.

4.4.5 Ein vollständiges Beispiel in VHDL

Um die vorgestellten Einzelheiten der Transformation noch einmal exemplarisch darzustellen, wurde die SSA SCL Version von ABO aus Abbildung 4.4 nach VHDL transformiert, Abbildung 4.38. In diesem Listing ist ebenfalls die transformierte Logik zu sehen. Die einzelnen Konstrukte wie Entity Deklaration, Sampling Register, Pre-Register und das GO-Register sind aufgeführt. Als Rücksetz- und Startvariante wurde die zweite Reset-Variante umgesetzt. Das Programm entspricht dem Verhalten des SCCharts aus Abbildung 4.4. Der synthetisierte Schaltkreis ist in Abbildung 4.39 gezeigt.

```
1  ENTITY ABO IS
2  PORT(
3      -- control
4       tick : IN std_logic;
5       Reset : IN boolean;
6       --inputs
7       A: IN boolean;
8       B: IN boolean;
9       -- outputs
10      A_out : OUT boolean;
11      B_out : OUT boolean;
12      O1 : OUT boolean;
13      O2 : OUT boolean
14     );
15 END ABO;
16
17 ARCHITECTURE behavior OF ABO IS
18 -- local signals definition, hided
19 begin
20     -- main logic
21     g0_1 <= GO_local;
22     O1_1 <= false WHEN g0_1 ELSE O1_pre;
23     O2_1 <= false WHEN g0_1 ELSE O2_pre;
24     g5_1 <= g4_pre;
25     g7_1 <= g6_pre;
26     g2_1 <= g0_1 or g5_1;
27     g3_1 <= g2_1 and A_local;
28     B_1 <= true WHEN g3_1 ELSE B_local;
29     O1_2 <= true WHEN g3_1 ELSE O1_1;
30     g4_1 <= g2_1 and not A_local;
31     g8_1 <= g7_1;
32     g9_1 <= g8_1 and B_1;
33     O1_3 <= true WHEN g9_1 ELSE O1_2;
34     g6_1 <= g0_1 or (g8_1 and not B_1);
35     e2_1 <= not (g4_1);
36     e6_1 <= not (g6_1);
37     g1_1 <= (g3_1 or e2_1) and
38          (g9_1 or e6_1) and
39          (g3_1 or g9_1);
40     O1_4 <= false WHEN g1_1 ELSE O1_3;
41     O2_2 <= true WHEN g1_1 ELSE O2_1;
42     -- Assign outputs
43     A_out <= A_local;
44     B_out <= B_1;
45     O1 <= O1_4;
46     O2 <= O2_2;
```

```
47     -- ---------------------
48     -- Pre-Registers
49     registers: process
50     begin
51     wait until rising_edge(tick);
52         if((Reset or Reset_local) = true) then
53             g4_pre <= false;
54             g6_pre <= false;
55             O1_pre <= false;
56             O2_pre <= false;
57         else
58             g4_pre <= g4_1;
59             g6_pre <= g6_1;
60             O1_pre <= O1_4;
61             O2_pre <= O2_2;
62         end if;
63     end process;
64
65    -- Sampling Register
66     inputRegister: process
67     begin
68     wait until rising_edge(tick);
69         if(reset = true) then
70             A_local <= false;
71             B_local <= false;
72         else
73             A_local <= A;
74             B_local <= B;
75         end if;
76     end process;
77
78    -- GO Register
79     GORegister: process
80     begin
81     wait until rising_edge(tick);
82       Reset_local <= Reset; -- has no reset input
83         if(reset = true) then
84             GO_local <= false;
85         else
86             GO_local <= Reset_local;
87         end if;
88     end process;
89 end behavior;
```

Abbildung 4.38. Vollständiger ABO VHDL Code — erzeugt mit der SSA SCL Transformation

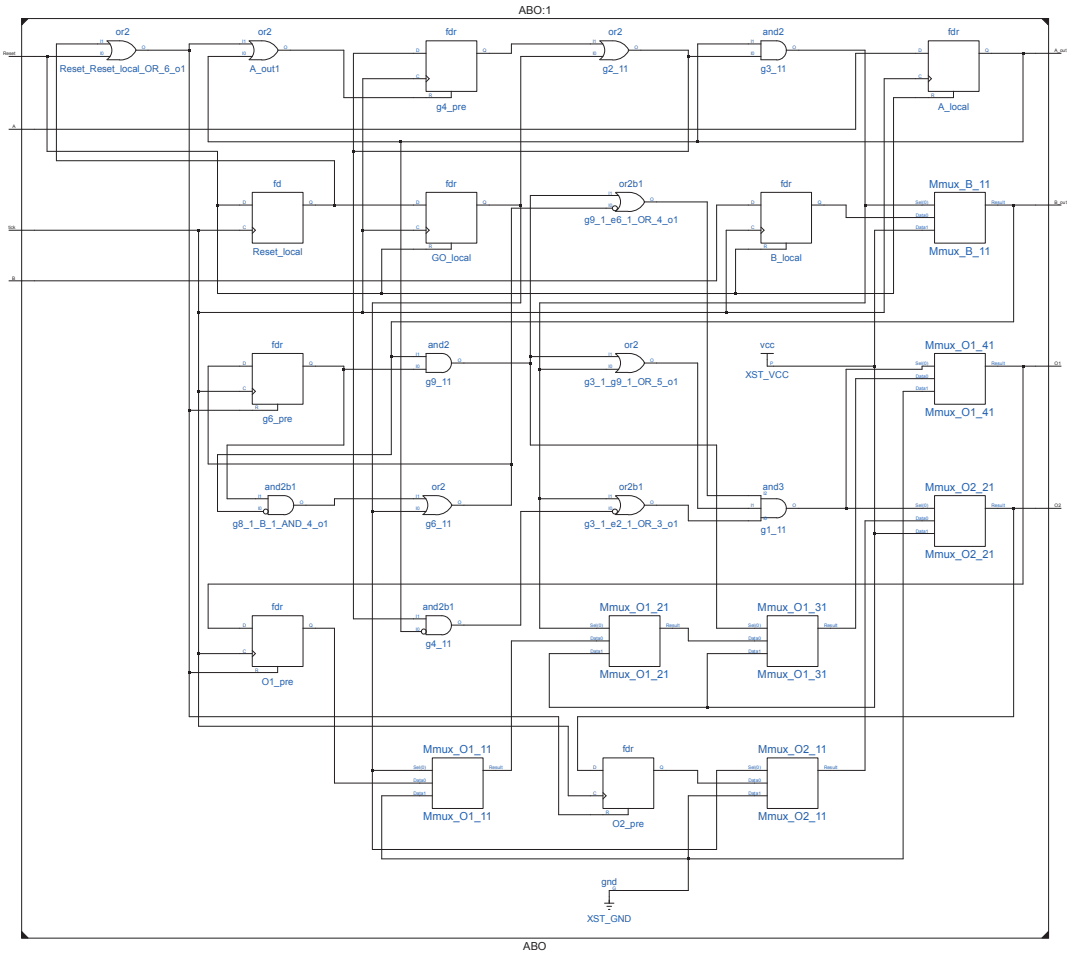


Abbildung 4.39. ABO Schaltplan generiert aus dem VHDL Code der SSA SCL Transformation

Kapitel 5

Automatischer Test

Die steigende Komplexität von Entwurfsmodellen und deren Transformationen kann beliebig kompliziert werden. Einfache Modelle können meistens durch den Programmierer kontrolliert werden. Bei größeren Modellen wird dies durch die Komplexität zunehmend schwieriger. Gerade bei der Transformation von SCCharts zu Hardware ist die manuelle Überprüfung sehr komplex. Die Validierung des transformierten VHDL-Codes oder auch des synthetisierten Schaltkreises stellt sich als aufwändig heraus. Aus dem Grund ist es wichtig, schon während der Entwicklung von Transformationen auf Tests zurückgreifen zu können, die sicherstellen, dass eine Transformation korrekt ist.

Damit die Korrektheit der Transformation validiert werden kann, wurde ein Testverfahren entworfen, welches die Funktion der transformierten Modelle testet. Der Test prüft das transformierte SCChart gegen eine Testdatei, welche ein vorgegebenes Verhalten des Modells beschreibt. In sogenannten ESO Dateien wird das Verhalten eines Modells durch die Definition der Ein- und Ausgabesignale in jedem Tick festgelegt. ESO Dateien werden unter anderem zum Testen von SyncCharts und extended SCCharts verwendet und können in KIELER oder textuell erstellt werden. Für den Test von SCCharts werden die ESO Dateien in eine VHDL-Form gebracht, mit dem der Schaltkreis getestet werden kann. Zum Testen wird das Xilinx Werkzeug ISim verwendet.

In diesem Kapitel wird das ESO Format erklärt, welches die Testsignale enthält. Bevor das Format verwendet werden kann, muss es in ein Zwischenformat, das Core ESO Format, transformiert werden. Aus dem Core ESO Format kann eine VHDL Repräsentation, die Testbench, generiert werden. Die Testbench kann zum Testen von VHDL-Dateien verwendet werden. Weiterhin wird dargelegt, wie man ein Modell gegen eine Testbench testen kann und wie die Ergebnisse interpretiert werden. Da Transformationen sehr komplex sein können, wird ein Testmodell selten alle Einzelheiten des Entwurfsmodells abdecken können, deshalb werden mehrere Modelle benötigt, um eine Transformation umfangreich testen zu können. Da es jedoch mühselig ist, nach jeder Änderung der Transformation alle Modelle zu testen, wird eine Lösung für einen automatischen Test präsentiert.

5.1 Testen vom Modellen

Das Testen und Simulieren von Modellen oder Editoren ist ein fester Bestandteil des Semantik Teilbereichs von KIELER. So stellt der KIELER Execution Manager (KIEM) die Möglichkeit bereit, Regressionstests [Mot09] während der Entwicklung durchzuführen. Bei der Entwicklung von extended SCCharts können ESO Dateien verwendet werden, die für jeden Tick ein eindeutiges Verhalten bzgl. der Ein- und Ausgangssignale eines Modells definieren. Das Modell wird gegen diese Definition getestet. Wenn alle erzeugten Ausgaben des Modells mit den erwarteten Ausgaben übereinstimmen, ist der Test bestanden. Die genannten Testdateien können im Core ESO Format für die Entwicklung der Hardwaresynthese aus SCCharts verwendet werden. Im Folgenden wird das ESO Format vorgestellt und die benötigte Zwischenrepräsentation Core ESO.

5.1.1 Das ESI/ESO Format

ESI/ESO steht für *Esterel Simulator Input* und *Esterel Simulator Output*. Mit dem ESI-Format werden Eingaben für einen Test eines Esterel Moduls definiert. In der ESO Datei sind nach dem Test die generierten Ausgaben vom Modell gespeichert. Mit Hilfe der beiden Dateien kann das Verhalten eines Esterel-Moduls validiert werden. In KIELER wird das ESI Format in einer erweiterten Form verwendet. Die beiden Dateien werden in dem ESI Format zusammengefasst, so dass Ein- und Ausgabesignale in einer Datei definiert werden können. Das Format entspricht weiterhin der ESI Grammatik, da es jedoch Ausgaben enthält, wird es hier als ESO Format bezeichnet. Das folgende Kapitel vermittelt die Idee vom KIELER ESO Format.

Das ESO Format

In dem KIELER ESO Format, im Folgenden als ESO Format bezeichnet, handelt es sich um eine Spezifizierung von Ein- und Ausgangssignalen für jeden Tick in einer Datei. In Listing 5.1 ist eine ESO Datei mit einem Testdurchlauf für das ABO-Beispiel aufgeführt.

```
1 !reset ;
2 % Output : ;
3 A
4 % Output : O2 A B ;
```

Listing 5.1. Ein ESO Trace für ABO

Das ESO Format spezifiziert welchen Wert ein Signal in einem Tick führt. Zu Beginn einer ESO Datei wird das Modell mit dem Befehl !reset zurückgesetzt. In einer Datei können mehrere Testdurchläufe, sogenannte *Traces*, angegeben werden, die durch die !reset-Anweisung getrennt sind. Das Semikolon beschreibt jeweils die Grenze eines Ticks. Zu Beginn eines Ticks werden die Werte der Eingabevariablen

definiert. Diese Werte werden durch die Simulationsumgebung an das zu testende Modell weitergegeben. Die erwarteten Ausgaben für ein Signal werden nach dem Schlüsselwort Output angegeben. Die Ausgaben sind als Kommentare (%) aufgeführt, damit das Format weiterhin der ESI/ESO Grammatik entspricht.

Aufgeführte Signale haben den Status present, nicht aufgeführte Signal sind absent. In einer ESO Datei können auch wertbehaftete Signale angegeben werden. Dabei handelt es sich um Signale, welche neben ihrem present oder absent Status noch einen Wert mit sich führen [Ber00]. In einer ESO Datei gibt es, aus Gründen der nichtredundanten Datenhaltung, keine Deklarationen der Ein- und Ausgaben des zu testenden Modells. Diese sind bereits in dem Modell selbst gespeichert.

5.1.2 Das Core ESO Format

Das ESO Format spezifiziert eine Testmöglichkeit, mit der das Verhalten eines extended SCCharts überprüft werden kann. In Extended SCCharts werden unter anderem Signale verwendet, wie im ESO Format. Core SCCharts hingegen verwenden nur Variablen. Das vorliegende ESO Format kann deshalb nicht als Testdatei für Core SCCharts verwendet werden. Damit das Format auch für Core SCCharts genutzt werden kann, wurde die ESO Grammatik um Variablen erweitert. Das Core ESO Format ist eine Teilmenge des ESO Formats. Es besteht somit die Möglichkeit auch in ESO Dateien Variablen anzulegen. Testdateien für Core SCCharts können in einem Core ESO Format angegeben werden. In Listing 5.2 ist eine Testdatei für ABO im ESO Format angegeben, welches nur Core Konstrukte verwendet. Die Datei beinhaltet einen Trace.

```
1  !reset ;
2  %% A = false
3  %% B = false
4  %% Output : %% A = false
5           %% B = false
6           %% O1 = false
7           %% O2 = false ;
8  %% A = true
9  %% B = false
10 %% Output : %% A = true
11          %% B = true
12          %% O1 = false
13          %% O2 = true ;
```

Listing 5.2. Core ESO Trace für ABO

Das Core ESO Format enthält keine Signale mehr, sondern nur noch Variablen. Eingabevariablen werden durch einen ESO Kommentar mit doppelten Prozentzeichen (%%) definiert, damit die ESI/ESO Syntax eingehalten wird. Ausgabevariablen werden durch ein doppeltes Prozentzeichen und das Schlüsselwort output angeführt. Variablen werden als Zuweisung mit Namen und Wert angegeben.

Alle Interface-Variablen, die im Modell verwendet werden, müssen im Core ESO in jedem Tick einen gültigen Wert besitzen. In dem ESO Format wurden Signale, die absent sind, nicht aufgeführt. Im Core ESO Format müssen die entsprechenden Variablen explizit auf false gesetzt werden, wenn sie nicht gesetzt sind.

5.1.3 ESO zu Core ESO Transformation

Um schon vorhandene extended SCChart Testdateien für Core SCCharts verwenden zu können oder Testdateien auf der Ebene des ESO Formats beschreiben zu können, wurde eine Transformation für ESO Dateien entwickelt, die beliebige ESO Dateien in eine äquivalente Core ESO Datei übersetzt. Das Listing 5.1 zeigt eine ESO Datei für das ABO Beispiel.

Dabei handelt es sich um eine M2M-Transformation auf dem ESO Modell. Core ESO Modelle lassen sich mit dem um Variablen erweiterten ESO Modell modellieren. Die Transformation kann relativ einfach gehalten werden. Es müssen für jeden existierenden Trace und für jeden darin enthaltenen Tick die Ein- und Ausgabesignale in Variablen transformiert werden. So ist im Listing 5.1 ein Trace mit zwei enthaltenen Ticks abgebildet. Signale können in boolesche Variablen abgebildet werden, wie in der Transformation von extended zu Core SCCharts dargelegt wurde, siehe Abschnitt 4.2.1. Hierbei wird der present-Status eines Signals, durch den Wert true und der absent-Status durch den Wert false dargestellt. Signale, die in der ESO Datei aufgeführt sind, werden emittiert, daher wird die entsprechende Variable in der Core ESO Datei auf true gesetzt. Nicht aufgeführte Signale sind absent. Da in einer ESO Datei nicht alle Signale bekannt sein müssen, kann nicht allein aufgrund dieser Datei festgestellt werden, welche Variablen in einem Tick auf false gesetzt werden müssen. Dazu wird die Deklarationsbeschreibung aus dem SCL-Modell benötigt. In der Deklaration sind alle Variablen aufgeführt. Alle Variablen, die in einem Tick noch nicht gesetzt wurden, können nun identifiziert und auf false gesetzt werden. Im Listing 5.2 ist die transformierte ESO Datei aus dem Listing 5.1 zusehen. Im ersten Tick der ESO Datei werden keine Eingabesignale emittiert, aus diesem Grund werden die Variablen der Core ESO Datei auf false gesetzt. Gleiches gilt für die Ausgabesignale der ESO Datei. Im zweiten Tick hingegen, werden für Ein- und Ausgangssignale Signale emittiert, die entsprechenden Variablen der Core ESO Datei werden daraufhin auf true gesetzt. Für Signale die in diesem Tick nicht vorhanden sind, wird den Variablen der Core ESO Datei der Wert false zugewiesen.

Diese Transformation unterstützt auch die Übersetzung von wertbehafteten Signalen, siehe Abbildung 5.1. Wertbehaftete Signale besitzen einen Status, der absent oder present sein kann, sowie einen dazugehörigen Wert. Signale mit Werten können in zwei Variablen aufgeteilt werden, bei dem eine Variable den Wert und die anderen den *present*-Status widerspiegelt [vHMA+13a]. Wobei die gleichnamige Variable den present-Status beinhaltet und eine Variable mit der Erweiterung $\_value$ den Wert. Es werden wertbehaftete Signale mit den Typen Boolean und Integer unterstützt. Wenn

```
1 !reset;
2 A B(5)
3 % Output : C D(false);
```

(a) ESO Trace Beispiel mit wertbahafteten Signalen

```
1 !reset ;
2 %% A = true
3 %% B_value = 5
4 %% B = true
5 %% Output :
6     %% C = true
7     %% D_value = false
8     %% D = true ;
```

(b) Core ESO Trace mit aufgelösten wertbehafteten Signalen in getrennte Variablen

Abbildung 5.1. Transformation eines ESO zu Core ESO Traces mit wertbehafteten Signalen

ein wertbehaftetes Signal absent ist, wird nur die Variable, die den Status hält, auf false gesetzt. Die Variable mit dem Wert bleibt unberührt.

Die Abbildung 5.1 zeigt eine Transformation einer ESO Datei zu einer Core ESO Datei. Es sind in der ESO Datei sowohl Signale als auch wertbehaftete Signale enthalten.

Implementierung

Wie im vorhergehenden Kapitel beschrieben, ist der Aufwand der ESO zu Core ESO Transformation überschaubar. Einfach gesagt, müssen alle Signale in Variablen transformiert werden und Signale eines Modells, die in der ES -Datei in einem Tick nicht angegeben sind, müssen in Variablen, die mit false beschrieben werden, transformiert werden. Die Transformation ist in dem Eclipse Plugin *de.cau.cs.kieler.eso.coreeso* eingebettet. Als Programmiersprache wurde Xtend gewählt. Core ESO Dateien haben die Dateiendung *.core.eso*.

Um die Transformation zu starten, wird in der laufenden Eclipse Instanz eine ESO Datei ausgewählt und ihr Kontextmenü aufgerufen. Im Menü erscheint der Eintrag Transform to Core ESO, welcher die Transformation ausführt. Der Handler des Events berechnet den Pfad der Modelldatei und ruft die Methode *transform-Eso2CoreEso* mit der gesamten ESO tracelist (sie enthält alle Testdurchläufe) und dem Modellpfad auf. Zu Beginn der Transformation muss das zugehörige SCL-Modell geladen werden. Damit zu einer ESO Datei das passende Modell geladen werden kann, muss es den gleichen Namen wie die Testdatei besitzen und im selben Ordner hinterlegt sein. Kann das Modell aus dem berechneten Pfad nicht geladen werden, wird die Transformation abgebrochen.

Aus dem Modell werden die Definitionen der Ein- und Ausgabevariablen geladen. Sie werden während der Transformation benötigt, um feststellen zu können, welche Variable in einem Tick der ESO Datei nicht aufgeführt sind. Die Methode *computeKVP* berechnet die Werte aller Variablen in einem Tick. Dazu werden der Methode die Interface Deklaration des Modells und die im aktuellen Tick aufgeführten Signale der

ESO Datei übergeben. Anhand der beiden Listen werden die Werte aller Variablen berechnet und eine Liste zurückgegeben. Variablen, die in dem Tick der ESO Datei angegeben sind, werden entsprechend gesetzt. Variablen, die nicht angegeben sind, werden auf false gesetzt.

Der Code in Listing 5.3 zeigt den Teil der ESO Transformation, der die ESO Traces in Core ESO Traces übersetzt. So werden für alle Traces und alle enthaltenen Ticks die Signale in einer neuen Tracelist in Variablen transformiert. Die aufgeführten *extraInfos* und *extraInfosOutput* Felder speichern die Variablen für Ein- und Ausgabevariablen.

```
// the new core eso tracelist
val coreTracelist = EsoFactory::eINSTANCE.createtracelist
// transform all traces
tracelist.traces.forEach[trace |

    // create a new trace
    newTrace = EsoFactory::eINSTANCE.createtrace

    // compute all ticks from current trace
    trace.ticks.forEach[tick |

        // generate a new tick
        newTick = EsoFactory::eINSTANCE.createtick

        // add all existing extraInfos to the new extraInfo field
        newTick.extraInfos.addAll(tick.extraInfos)

        // transform all input variables from the current tick
        newTick.extraInfos.addAll(computeKVP(tick.input, inputDefinitions))

        // add all existing extraInfosOutput to the new extraInfoOutput field
        newTick.extraInfosOutput.addAll(tick.extraInfosOutput)

        // transform all outputs variables from the current tick
        newTick.extraInfosOutput.addAll(computeKVP(tick.output, outputDefinitions))

        // add new generated tick to new trace
        newTrace.ticks.add(newTick)
    ]
    // add new generated trace to the trace list
    coreTracelist.traces.add(newTrace)
]
// return new core tracelist
return coreTracelist
```

Listing 5.3. Pseudo Code der Methode zum Generieren von Core ESO Dateien aus ESO Dateien

5.1.4 ESO zu VHDL Testbench Transformation

Die vorliegenden Core ESO Dateien könnten zum Testen von Core SCCharts verwendet werden. Es ist jedoch noch nicht möglich, die Testdateien zum Testen des

VHDL-Codes zu verwenden. Dazu muss die ESO Datei in eine VHDL-Testbench übersetzt werden. Mit einer Testbench können VHDL-Komponenten bezüglich ihres Verhaltens überprüft werden. Zum Überprüfen eignet sich der ISE Simulator (ISim). Er ist zum Testen, Analysieren und Simulieren von VHDL Entities entwickelt worden. Eine vorliegende Testbench kann zusammen mit dem transformierten Modell im ISim geladen werden. ISim kann die Komponente entsprechend der gegebenen Testbench überprüfen und eine Auskunft darüber geben, ob sich das Programm entsprechend der Erwartungen, die in der Testbench festgelegt wurden, verhält. Im folgenden Verlauf wird die Funktion einer Testbench erläutert, sowie auf die Transformation eingegangen.

Die VHDL Testbench

Bei einer Testbench handelt es sich um eine textuelle VHDL-Beschreibung, mit der individuelle VHDL-Komponenten, mit definierten Eingangssignalen getestet werden können. Das zu testende Modul, bezeichnet als Unit Under Test (UUT), wird in der Testbench instantiiert und kann über sein Interface angesteuert werden. Es können definierte Signale an die Eingabeports des UUT gegeben und die Ausgaben gelesen werden. Das Verhalten und die Korrektheit können somit überprüft werden. Eine schematische Darstellung einer Testbench und die dazugehörige Beschreibung ist in Abschnitt 3.4.2 aufgeführt. Die Signalverläufe des Tests können in Form eines Signaldiagramms ausgewertet werden.

Eine Testbench simuliert Signale gegenüber dem UUT und ist nicht synthesefähig, das bedeutet, dass die Testbench nicht auf einen FPGA konfiguriert werden kann. Sie ist selbst eine Entity ohne Ein- und Ausgänge. Die VHDL-Syntax steht mit zusätzlichen Befehlen zur Programmierung zur Verfügung. Das Validieren von VHDL-Modellen mit Testbenches hat sich als Standardtestverfahren etabliert. Eine Testbench kann in vier Abschnitte eingeteilt werden. Eine Darstellung der Abschnitte ist in Listing 5.4 zu sehen.

Entity und Architektur Beschreibung: Die Entity- und Architekturbeschreibung stellt das Grundgerüst einer Testbench dar. Die Testbench selber besitzt keine Ein- oder Ausgänge (Zeile 2). In der Architekturbeschreibung wird das Verhalten der Testbench programmiert.

Signal Deklaration: In der Signal Deklaration (Zeile 12) werden lokal verwendete Signale deklariert. So wird unter anderem das Taktsignal, welches in der Regel benötigt wird, deklariert.

UUT Instantiierung: Das UUT, welches mit der Testbench getestet werden soll, muss bekannt gemacht werden. Dazu wird die zu testende Top-Level-Entity als eine Komponente in der Testbench angelegt (Zeile 8) und instantiiert (Zeile 16). Lokal

```
1  -- Entity has no In- or Outputs
2  ENTITY abo_tb IS
3  END abo_tb;
4
5  ARCHITECTURE behavior OF abo_tb IS
6
7  -- component declaration
8  COMPONENT abo
9  -- port declaration
10 END COMPONENT;
11
12 -- local Signal Declaration
13
14 BEGIN
15
16     uut: UUT_NAME(
17    --map port signals to local signals
18     );
19
20 -- generate Input Signals
21
22 END;
```

Listing 5.4. VHDL Testbench — Struktureller Aufbau

deklarierte Signale werden bei Instantiierung mit der Komponenten verbunden. So können Eingangssignale an die UUT gegeben und Ausgaben gelesen werden.

Definition der Eingangssignale: Als letztes werden die Eingangssignale definiert, mit denen das UUT getestet werden soll (Zeile 20). Des Weiteren können *Assertions* für Ausgangssignale des UUT angelegt werden. *Assertions* testen Signale darauf, ob sie zu einer bestimmten Zeit einen definierten Wert besitzen. Führt das Signal einen anderen als den angegebenen Wert, wird ein Fehler ausgelöst.

Simulation des Tick-Signals In einer Testbench können auch periodische Signale definiert werden, die parallel zu anderen Signalen generiert werden. Häufig werden auf diese Art Taktsignale erzeugt. Für Core SCCharts kann auf diese Weise das Tick-Signal erzeugt werden.

Transformation von Core ESO zu einer VHDL Testbench

Damit eine vollständige Testbench aus einer Core-ESO Datei generiert werden kann, müssen die vier Abschnitte einer Testbench generiert werden. Für diese Transformation werden die Core ESO Datei sowie das zugehörige SCL-Modell benötigt. Das Modell wird benötigt, um die Interface Deklaration des UUT in der Testbench zu erzeugen. Die Transformation der ESO Testdatei aus Listing 5.2 ist in Abbildung 5.2 gezeigt und wird für die folgenden Erklärungen verwendet.

```vhdl
ENTITY ABO_tb IS
END ABO_tb;

ARCHITECTURE behavior OF ABO_tb IS

COMPONENT ABO
PORT(
      -- control
      tick : IN std_logic;
      reset : IN boolean;
      -- inputs
      A: IN boolean;
      B: IN boolean;
      -- outputs
      A_out : OUT boolean;
      B_out : OUT boolean;
      O1 : OUT boolean;
      O2 : OUT boolean
    );
END COMPONENT;

-- Inputs
signal A : boolean := false;
signal B : boolean := false;
-- Outputs
signal A_out : boolean := false;
signal B_out : boolean := false;
signal O1 : boolean := false;
signal O2 : boolean := false;
-- Control
signal reset : boolean := false;
signal tick : std_logic := '0';
constant tick_period : time := 100 ns;

BEGIN
   uut: ABO PORT MAP(
      tick => tick,
      reset => reset,
      -- Inputs
      A => A,
      B => B,
      -- Outputs
      A_out => A_out,
      B_out => B_out,
      O1 => O1,
      O2 => O2);

   tick_process: process
   begin
      tick <= '0';
      wait for tick_period/2;
      tick <= '1';
      wait for tick_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      wait for 1 ps;
      -- NEW TRACE
      reset <= true;
      wait for tick_period;
      reset <= false;

      -- tick 1
      A <= false;
      B <= false;
      wait for tick_period;
      assert( A_out = false )
         report "1st trace: 1st tick:
                 A_out should have been false"
         severity ERROR;
      assert( B_out = false )
         report "1st trace: 1st tick:
              B_out should have been false"
         severity ERROR;
      assert( O1 = false )
         report "1st trace: 1st tick:
              O1 should have been false"
         severity ERROR;
      assert( O2 = false )
         report "1st trace: 1st tick:
              O2 should have been false"
         severity ERROR;

      -- tick 2
      A <= true;
      B <= false;
      wait for tick_period;
      assert( A_out = true )
         report "1st trace: 2nd tick:
              A_out should have been true"
         severity ERROR;
      assert( B_out = true )
         report "1st trace: 2nd tick:
              B_out should have been true"
         severity ERROR;
      assert( O1 = false )
         report "1st trace: 2nd tick:
              O1 should have been false"
         severity ERROR;
      assert( O2 = true )
         report "1st trace: 2nd tick:
              O2 should have been true"
         severity ERROR;
      wait;
   end process;
END;
```

Abbildung 5.2. Vollständige ABO VHDL Testbench

Die Transformation kann in einem Durchlauf aus der Core ESO Datei eine VHDL Testbench erzeugen. Dazu wird zu Beginn das SCL Modell geladen, um aus dem Modell die Interface Deklaration zu extrahieren. Im ersten Schritt wird aus dem Namen der SCL Datei der Name der Testbench Entity generiert (Zeile 1), indem an den Namen die Erweiterung *_tb* für Testbench gehängt wird. In der Architekturbeschreibung wird eine VHDL Komponente des SCL-Modells angelegt (ab Zeile 6). Diese Komponente muss der Entity Deklaration aus der generierten VHDL Datei vom SCChart entsprechen, da diese später instantiiert und getestet wird. Die Beschreibungen stimmen überein, da sie aus demselben SCL-Modell generiert werden.

Für jedes Ein- bzw. Ausgangssignal des UUT wird jeweils ein lokales Signal angelegt (ab Zeile 23). Die lokalen Signale werden innerhalb der Testbench genutzt, um die vorgegebenen Werte aus der ESO Datei an die zu testende Komponente zu führen und die produzierten Werte zu lesen und zu vergleichen. Die lokalen Signale haben dabei dieselben Namen wie die Signale der Komponente. Es wird weiterhin eine Konstante tick_period deklariert, sie bestimmt die Dauer eines Ticks. Die Dauer eines Ticks in der Simulation beträgt 100ns, siehe Abschnitt 4.4.3.

Innerhalb der Architekturbeschreibung nach der begin-Anweisung, wird zu Beginn das UUT instantiiert, wobei die Ein- und Ausgänge des UUT mit den gleichnamigen lokalen Signalen verbunden werden (ab Zeile 36).

Wie oben beschrieben, muss das Tick-Signal simuliert werden. Es wird dafür ein Prozess angelegt, welcher ein periodisches Tick-Signal generiert. Der Prozess heißt in der Abbildung 5.2 tick_process. Es wird ein periodische Tick-Signal erzeugt. Dieser Prozess läuft parallel zu den anderen Anweisungen und startet automatisch, da er keine Sensitivitätsliste bzw. wait until-Anweisung besitzt. Diese fehlenden Angaben sind nur in einer Testbench erlaubt. Die Ausführung wird erst beendet, wenn die Simulation beendet wird. Die bis zu diesem Punkt ausgeführte Transformation wurde mit den Informationen aus dem SCL Modell sowie statisch (Tick-Prozess) erstellt.

Der Prozess stim_proc wird mit Informationen aus der ESO Datei generiert. Er setzt die Eingangssignale und überprüft die Werte der Ausgangssignale, die in der ESO Datei angegeben sind. Da Testbenches auch zum Testen von Teilkomponenten eines größeren Systems genutzt werden können, muss eine Besonderheit beachtet werden, damit sich die Simulation korrekt verhält. Wenn in der Testbench Signale einer äußeren Komponenten simuliert werden sollen, die das eigentliche UUT umgibt, ist es wichtig, dass die simulierten Signale zeitlich nach dem Takt (tick) generiert werden. Ein kleines Beispiel: Angenommen, die Eingangssignale werden von einem getakteten Register einer äußeren Stufe generiert. Die Register übernehmen den Wert an ihrem Eingang mit der steigenden Taktflanke und stellen den gelesenen Wert nach einer Zeit δ erst an ihrem Ausgang zur Verfügung. Damit in der Testbench das simulierte Signal ebenfalls nicht synchron zum Taktsignal erzeugt wird, muss die Ausführung des stim_proc gegenüber dem tick_process verzögert werden. Der Prozess beginnt deshalb mit der Anweisung wait for 1ps, der den Simulationsprozess um eine

pico Sekunde verzögert. Eine pico Sekunde ist als Zeitversatz für die Simulation ausreichend, um das beschriebene Verhalten zu erreichen.

Um das zu testende Modell zu initialisieren, wird der Reset aus der ESO Datei ebenfalls in VHDL übersetzt. Für des Rücksetzen wird, wie im Abschnitt 4.4.4 ausgeführt, ein eigener Takt verwendet. Zum Rücksetzen wird das Signal reset für die Dauer eines Tick gesetzt (Zeile 60-62).

Nun folgt die eigentliche Simulation der Eingangssignale. Sie wird entsprechend der ESO Datei generiert. Dazu werden zu Beginn eines jeden Ticks alle Eingangssignale entsprechend des aktuellen Ticks in der ESO Datei gesetzt (Zeilen 65,66,86,87). Nachdem der synthetisierte Schaltkreis eine halbe Tick-Periode Zeit zum Rechnen bekommen hat (Zeilen 67,88), müssen die generierten Ausgangssignale überprüft werden. Im Quellcode wird eine ganze Tick-Periode gewartet (Zeilen 67,88). Da das Tick-Signal jedoch mit Low startet, wird die Ausgabe eine halbe Tick-Periode, nachdem der Tick gestartet wurde, getestet. Dazu werden *Assertions* verwendet. Für jeden Ausgang, den die Komponente besitzt, wird in jedem Tick eine Assertion angelegt (Zeilen 68-83,89-104). Die Assertion prüft den Wert, der in der Core ESO Datei, in dem aktuellen Tick, angegeben ist. Wenn eine Assertion verletzt wird, wird eine Fehlermeldung ausgegeben. Sie enthält Informationen darüber, in welchem Trace und in welchem Tick welcher Wert erwartet wurde. Diese Informationen können während der Transformation errechnet und eingefügt werden. Der severity-Wert ERROR (z.B.: Zeile 71) gibt an, dass ein Fehler aufgetreten ist, die Simulation aber nicht abgebrochen werden soll. Der Wert *FAILURE* würde die Simulation zum Beispiel beenden.

Implementierung

Die Implementierung zur vorgestellten Transformation kann in den Schritten umgesetzt werden, wie sie vorgestellt wurde. Die Transformation wurde mit Xtend programmiert und ist in dem Plugin *de.cau.cs.kieler.eso.vhdl* zu finden. Für VHDL, einschließlich des erweiterten Befehlssatzes für die Testbench, gibt es kein Meta-Modell, weswegen hier eine Modell-zu-Text Transformation vorliegt. Es werden strukturierte und lesbare VHDL-Dateien erzeugt. Die Rich Strings von Xtend bieten die Möglichkeit zur strukturierten Codeerzeugung, siehe Abschnitt 3.1.2.

Die Transformation von einer Core ESO Datei zu einer VHDL Testbench kann über das Kontextmenü der Core ESO Datei ausgerufen werden. Wenn die Transformation angestoßen wird, muss zu Beginn der Pfad des SCL-Modells errechnet und der Transformationsmethode zusammen mit der Tracelist aus der Core ESO Datei übergeben werden. Die Transformationsmethode *transformESO2VHDL* versucht zu Beginn aus dem übergebenen Pfad das SCL-Modell zu laden. Das Modell muss in den selben Ordner wie die ESO Datei liegen und sie muss denselben Namen besitzen. Die Dateien werden anhand ihrer Endungen (.scl und .core.eso) unterschieden. Es wird versucht, das Modell zu laden, in dem der Name der ESO Datei genommen wird

```
'''«/* Entity declaration from testbench entity */»
ENTITY «entityName»_tb IS
END «entityName»_tb;

ARCHITECTURE behavior OF «entityName»_tb IS
«/* Generate component port declaration from the to tested VHDL component  */»
«generateComponent(modelInputs, modelOutputs, entityName)»

--Inputs«/* Declare local signals for all input signals */»
«modelInputs.map( in |'''«generateVhdlSignalFromVariableWithInitialValue(in,"")»''').join('\n')»

--Outputs«/* Declare local signals for all output signals*/»
«modelOutputs.map( out |'''«generateVhdlSignalFromVariableWithInitialValue(out,"")»''').join('\n')»

--Control«/* Generate control signals, every component need a tick and reset pin */»
signal reset : boolean := false;
signal tick : std_logic := '0';
signal term : boolean := false;
constant tick_period : time := 100 ns;

BEGIN«/* fill testbench with functionality */»
    «/* instantiate the Unit Under Test (seq-SCL-VHDL-model)  */»
    «generateUUT(modelInputs, modelOutputs, entityName)»

    «/* The testbench simulates the tick, generate that process */»
    «generateTickProcess()»

    «/* Here the simulation process is created */»
    «generateSimulation(esoTracelist, modelInputs, modelOutputs)»

END;
'''
```

Abbildung 5.3. Xtend Quellcodeausschnitt der Transformationsmethode zum Erzeugen der Testbench Entity mit Hilfe von Rich Strings

und die Endung durch .scl ersetzt wird. Wenn das SCL-Modell nicht vorhanden ist, wird die Transformation abgebrochen. Der Name der Dateien bestimmt gleichzeitig den Namen der Testbench Entity. Der Name darf keine Bindestriche oder führenden Ziffern beinhalten.

Aus dem SCL-Modell wird die Interface Deklarationen extrahiert und die entsprechenden Ein-, Aus- und Ein-Ausgabevariablen ermittelt. In der Core ESO Datei können noch Ein-Ausgabevariablen vorhanden sein, die mit Hilfe der ermittelten Variablen aus dem SCL-Modell zugeordnet werden können. Da Ein-Ausgabevariablen in VHDL nicht verwendet werden können, siehe Abschnitt 4.4.3, müssen sie transformiert werden. Dazu werden in allen Core ESO Traces und allen Ticks alle Ausgabevariablen durchsucht. Wenn eine Variable im SCL-Modell als Ein-Ausgabevariable definiert ist, wird die Erweiterung *_out* an den jeweilige Namen der Ausgabevariable in der ESO Datei gehängt. Der Name der Variablen stimmt daraufhin mit der VHDL Interface Beschreibung des Modells überein, dort wurden Ein-Ausgabevariablen nach demselben Schema aufgeteilt, siehe Abschnitt 4.4.3.

Nachdem die Vorarbeiten erledigt sind, kann die Generierung der Testbench starten. Das Listing 5.3 zeigt eine Ausschnitt aus der Methode *createEntity*, in dem die Entity mittels Rich Strings generiert wird.

```
def generateTickProcess() {
'''
tick_process: process
begin
    tick <= '0';
    wait for tick_period/2;
    tick <= '1';
    wait for tick_period/2;
end process;
'''
}
```

Abbildung 5.4. Xtend Methode die den Tick Prozess mit Hilfe vom Rich Strings erzeugt

Im Codeausschnitt sind blau geschriebene Wörter vorhanden, unter anderem handelt es sich um VHDL-Schlüsselwörter, wie zum Beispiel ENTITY und ARCHITECTURE. Die blau geschriebenen Wörter sind Teile des Strings, der später den VHDL Code darstellt. Rich String werden mit ‴ umschlossen und grau hinterlegte Ausschnitte werden später als generierter String zurückgeliefert. In einem Rich String können Funktionen beliebiger Art aufgerufen werden, sie werden mit ≪≫ gekennzeichnet. Es kann zum Beispiel auf Variablen oder Funktionen zugegriffen werden. Der Rückgabewert muss ein String sein, der dem Rich String hinzugefügt wird. Innerhalb der BEGIN-Anweisung sind hinterlegte Einrückungen zu sehen, diese werden mit in den generierten Quellcode übernommen und dienen der Strukturierung.

Die aufgerufenen Methoden generieren jeweils die Teile des VHDL Codes, die ihr Name beschreibt. So generiert die Methode generateTickProcess den VHDL Code, der das Tick-Signal generiert, siehe Listing 5.4 und generateSimulation erstellt aus den Core ESO Traces die Eingangssignale und die Überprüfung der Ausgangssignale.

5.2 ISE Simulator

Nachdem die Transformationen des SCCharts in eine VHDL-Datei und der ESO Testdatei in eine VHDL Testbench beschrieben wurden, wird nun eine Möglichkeit vorgestellt, die beiden Dateien gegeneinander zu validieren. Das Xilinx Tool ISim stellt die Funktionen bereit, um eine VHDL Komponente mit einer Testbench zu testen. In der Entwicklungsumgebung ISE kann eine Testbench für beliebige Entities angelegt werden. Nachdem die Testbench für ein Modell geschrieben wurde, kann ISim aus ISE heraus gestartet werden.

Wenn ISim gestartet wird, werden alle Dateien, die zur testenden Entity gehören und die Testbench selbst kompiliert. Wenn keine Fehler auftreten, wird die Simulation der Testbench gestartet. ISim führt die Simulation aus und schreibt Ereignisse sowie verletzte Assertions in eine Log-Datei, siehe Listing 5.5. In Zeile 10 der Log-Datei ist eine Fehlermeldung einer Assertion aufgeführt. Die Log-Datei kann im Simulator angezeigt und ausgewertet werden. Für die Generierung der Log-Datei, muss nicht

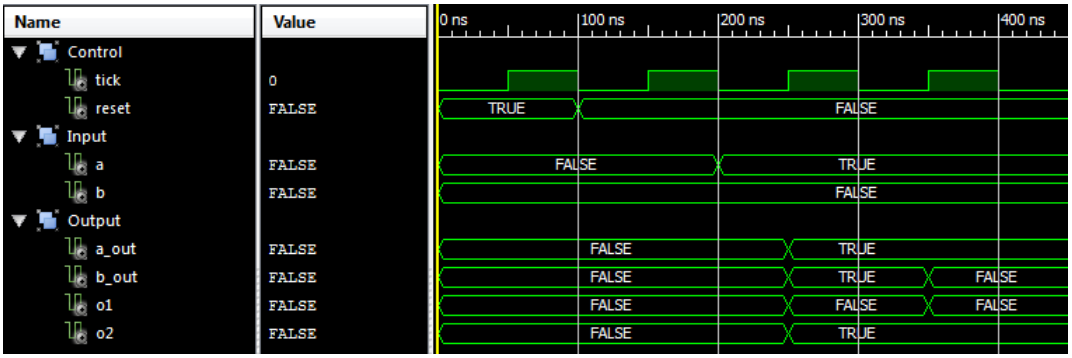


Abbildung 5.5. Signalverlauf von ABO im ISE Simulator

die gesamte grafische Oberfläche gestartet werden. ISim stellt eine Kommandozeilen-Schnittstelle bereit, über die das Tool bedient werden kann.

```
1  ISim log file
2  Running: c:\Users\gjohannsen\junit-workspace\temp-scl\ABO\tb_ABO_isim_beh.exe
3           -intstyle ise -tclbatch ABO.cmd -log out.log -sdfnowarn
4  ISim P.58f (signature 0x8ef4fb42)
5  This is a Full version of ISim.
6  Time resolution is 1 ps
7  # run 2700 ns
8  Simulator is doing circuit initialization process.
9  Finished circuit initialization process.
10 at 700001 ps: Error: 2nd trace: 2nd tick: B_out should have been true
11 # quit
```

Listing 5.5. Eine Log Datei vom ISE Simulator

Die Auswertung großer Log-Dateien gestaltet sich schnell als schwierig und unübersichtlich. Ebenso können Signalzusammenhänge nicht analysiert und Fehleranalysen mit Hilfe der Log-Dateien nicht gemacht werden. Deshalb bietet ISim die Möglichkeit, Signalverläufe grafisch anzuzeigen. Es können Signalverläufe kontrolliert und miteinander verglichen werden und die korrekte Funktionsweise überprüft werden. In Abbildung 5.5 ist der Signalverlauf der Simulation von ABO gezeigt. Es wurde die in Abbildung 5.2 vorgestellte Testbench verwendet. Die ersten beiden Signale sind tick und reset. Das Tick-Signal ist periodisch und ein Tick beginnt mit der steigenden Flanke. Der erste Tick beginnt mit der ersten Taktflanke bei 50ns. Im ersten Tick wird der Reset ausgeführt. Der Reset ist mit der steigenden Taktflanke bei 50ns gültig und wird ausgeführt (taktsynchroner Reset). Danach folgen die Eingangssignale a und b. Diese werden entsprechend der Testbench in jedem Tick beschrieben. Die Ausgangssignale a_out, b_out, o1 und o2 werden von der UUT (ABO) generiert. Der Signalverlauf zeigt entsprechend das in der ESO Datei definierte Verhalten, siehe Abbildung 5.2.

5.3 Regressionstest

Für ausführliche Tests von Transformationen sind im Normalfall mehrere verschiedene Modelle nötig. Mit steigender Komplexität der Modellierungssprache steigt auch der Aufwand, der für das Testen nötig ist. So müssen nach jeder größeren Änderung alle Modelle getestet und deren Ergebnisse ausgewertet werden, um zu überprüfen, ob die Transformation noch vollständig korrekt funktioniert.

Für diese Arbeit würde das bedeuten, dass jedes SCChart sowie eine entsprechende ESO Datei in ihre jeweilige VHDL-Repräsentation übersetzt werden müssten und anschließend ein Projekt im ISE Tool angelegt werden müsste, um die Dateien zu kompilieren. Schlägt der Kompiliervorgang fehl, liegt ein Fehler in der Transformation vor. Bei einer erfolgreichen Kompilierung würde der ISE Simulator gestartet werden und die Simulation würde das Modell testen. Nach dem Test muss die Log-Datei analysiert werden, damit das Ergebnis der Simulation festgestellt werden kann. Diese Prozedur müsste mit jedem Testmodell durchgeführt werden, um die Korrektheit der Transformation zu überprüfen.

Da diese Art zu testen sehr aufwändig ist, wurde für die Entwicklung ein automatischer Test entwickelt, der die vorgestellten Schritte automatisch ausführt und eine Rückmeldung über jeden Test liefert. Der Test wird einmal gestartet und für alle definierten Modelle durchgeführt. Im Folgenden werden der automatische Test sowie die Implementierung vorgestellt.

5.3.1 Funktionsweise des automatischen Regressionstests

Für den Test werden immer eine Modelldatei und die dazugehörige ESO Datei getestet und ausgewertet. In Abbildung 5.6 ist der Verlauf des Test grafisch dargestellt. Anhand der Grafik werden im Nachfolgenden die einzelnen Schritte des Tests erklärt.

Zu Beginn des Tests werden die zu testenden Komponenten geladen. Dazu müssen die Modelle, hier die SCCharts und die ESO Dateien, in einem vorher festgelegten Verzeichnis hinterlegt sein. Damit eine Modelldatei und die dazugehörige Testdatei identifiziert werden können, müssen Modell- und Testdatei denselben Namen besitzen. Nachdem alle Modelle und ihre zugehörigen Testdateien identifiziert wurden, wird für jedes dieser Paare der Test ausgeführt.

Als erstes wird das SCChart mit Hilfe der Transformationen von Smyth [Smy13] in SCL und danach in sequentielles SCL transformiert. Das sequentielle SCL Modell wird zuerst in ein SSA SCL Modell transformiert, bevor es zum Schluss in eine VHDL-Datei übersetzt wird. Die VHDL Datei wird im Dateisystem des Rechners hinterlegt.

Die zugehörige ESO Testdatei muss in eine VHDL-Testbench überführt werden. Dazu wird die ESO Datei in die Core Version transformiert und anschließend in eine Testbench. Die Testbench wird ebenfalls auf dem Rechner gespeichert.

Nun liegen Modell- und Testdatei in ihrer jeweiligen VHDL Version vor und können mit dem ISim simuliert werden. Die Xilinx Tools bieten die Möglichkeit, über die

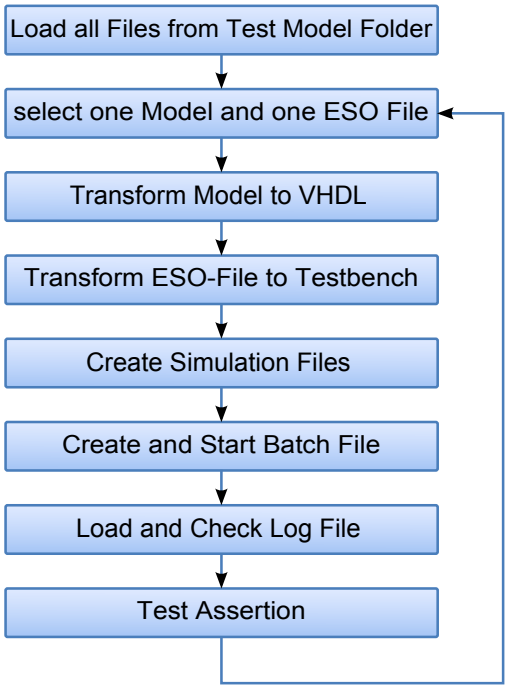


Abbildung 5.6. Die einzelnen Schritte des Regressionstests

Kommandozeile bedient zu werden. Es entfällt somit das manuelle Starten und Einrichten von Projekten in ISE und die Simulation in ISim. Damit die Xilinx Tools über die Kommandozeile ausgeführt werden können, sind weitere Informationen nötig, die sich sonst aus den angelegten Projekten und Einstellungen in den Programmen ergeben. Eine Top-Level-Entity kann aus verschiedenen einzelnen Entitys bestehen, die zusätzlich auf mehrere Dateien verteilt sein können. Es wird eine Projektdatei benötigt, die alle relevanten Dateien zusammenfasst, die für die zu testende Top-Level-Entity nötig sind. Für eine Simulation muss zusätzlich die Testbench zur Projektdatei hinzugefügt werden. Alle in der Projektdatei aufgeführten Dateien werden später mit dem ISE Compiler kompiliert. Für die Simulation wird eine *Command*-Datei benötigt, die Eigenschaften für die Simulation definiert, siehe Abschnitt 5.3.2. In der Command-Datei können neben der Simulationsdauer auch Signale definiert werden, die in dem *Waveform*-Editor angezeigt werden können. Da für diese Simulation keine Signale angezeigt werden sollen, muss nur die Dauer der Simulation angegeben werden. Die Dauer der Simulation ergibt sich aus der Anzahl der Ticks, die in der ESO Datei spezifiziert sind und aus der Dauer eines Ticks. Die kompilierte Datei des ISE Compilers sowie die Command-Datei werden später an den ISim übergeben.

Damit der erwähnte Compiler und die Simulation ausgeführt werden, wird eine Skriptdatei erzeugt, die diese Aufgabe übernimmt. In dem Skript wird der ISE Compiler zusammen mit der Projektdatei ausgeführt. Die entstandene Binärdatei

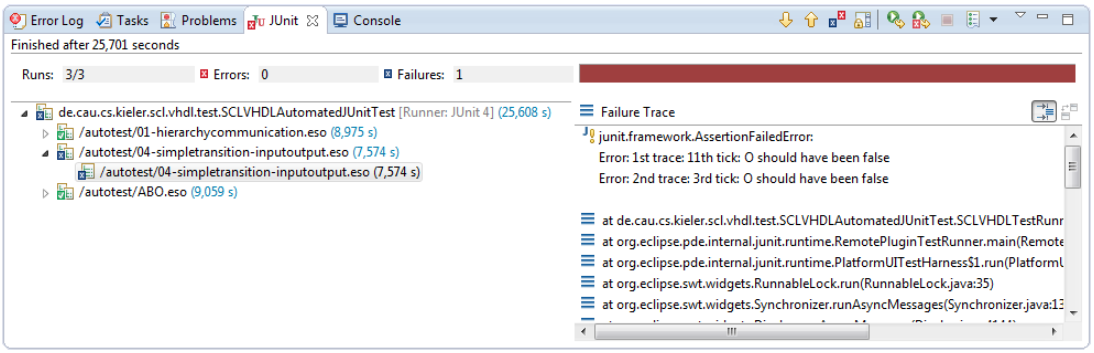


Abbildung 5.7. Ergebnisse eines Regressionstests

wird zusammen mit der Command-Datei im ISE Simulator getestet. Der Simulator beginnt mit dem Test der Dateien und erstellt eine Log-Datei mit den Ergebnissen der Simulation.

Diese Log-Datei wird wiederum von dem Test geöffnet und analysiert. Befinden sich in der Datei Einträge, die auf einen Fehler hinweisen, war die Simulation nicht erfolgreich. Diese Fehler werden im *Stacktrace* des jeweiligen Modells angezeigt. Die eventuell vorhandenen Fehlermeldungen enthalten die Informationen, die in der Testbench innerhalb der Assertions angegeben wurden. Somit kann der Trace und der Tick ermittelt werden, in dem der Fehler aufgetreten ist.

Der Ablauf wird für alle Modelle durchgeführt. Das in Abbildung 5.7 aufgeführte Protokoll zeigt einige bestandene und fehlgeschlagene Tests. Der Stacktrace zum jeweiligen Test enthält genaue Informationen zum Fehler.

Es können weitere Modelle zum Test hinzugefügt werden, indem sie in den entsprechenden Ordner eingefügt werden. Es werden alle Modelle in diesem Ordner getestet, zu denen eine entsprechende ESO Datei gefunden werden kann.

5.3.2 Implementierung

In diesem Teil der Arbeit werden einige Implementierungsdetails des automatischen Tests vorgestellt. Der automatische Test wurde mit Hilfe des JUnit Testframeworks in einem Plugin mit dem Namen *de.cau.cs.kieler.vhdl.test* implementiert. Wenn der Test über die Eclipseumgebung gestartet wird, startet eine neue Eclipse Instanz, die für den JUnit Test verwendet wird. Dazu verknüpft die Initialisierungsroutine *SCLVHDLTestRunnerInitialization* alle Dateien, die im Testordner angeben sind, mit diesem Workspace. Bei dem Testordner handelt es sich um einen Ordner, der in der Klasse definiert ist, in dem die Modelle und die Testdateien hinterlegt werden können. Nachdem alle Dateien mit dem Workspace verlinkt wurden, werden Paare aus SCChart und ESO Testdatei ermittelt und in einer Hashmap gespeichert. Kann für ein SCChart keine passende ESO Datei gefunden werden, wird sie nicht in die

Hashmap eingetragen. SCChart und Testdatei werden anhand der gleichen Namen identifiziert.

Der JUnit Test wird anschließend für jede Testdatei, zu der eine Modelldatei gefunden wurde, gestartet. Es wird die Methode *SCLVHDLTestRunnerExecution* ausgeführt. Die Methode führt die im letzten Kapitel genannten Schritte aus, indem die ESO Datei in eine Testbench und das SCChart in eine VHDL-Datei transformiert werden. Für jeden Test wird in dem Workspace ein eigener Ordner mit dem Namen des Modells angelegt, in dem die generierten Dateien gespeichert werden.

Im nächsten Schritt werden die Projekt- und die Command-Datei sowie die Skriptdatei generiert. Details zu dieser Generierung werden im nächsten Abschnitt erläutert. Diese Dateien werden ebenfalls in dem generierten Ordner gespeichert. Nachdem alle benötigten Dateien erstellt wurden, wird das Skript ausgeführt, welches den Compiler und die Simulation ausführt. Im letzten Schritt wird die vom Simulator erstellte Log-Datei analysiert. Näheres wird im folgenden Kapitel beschrieben. Bei der Analyse stellt sich heraus, ob die Simulation erfolgreich war. In dem Fall, dass die Simulation fehlschlägt, wird eine JUnit Exception generiert, die dem Framework das Fehlschlagen des Tests mitteilt. Im Folgenden werden noch weitere Eigenschaften zu den benötigten Dateien für den Compiler und Simulator vorgestellt, ebenso wie die Skriptdatei.

Projektdatei

Die Projektdatei wird für den ISE Compiler benötigt. Sie muss alle Dateien enthalten, die zur entsprechenden Top-Level-Entity gehören, die kompiliert werden sollen. Sie können in beliebiger Reihenfolge in der Datei aufgeführt sein. Die Testbench muss für den Test ebenfalls mit aufgeführt sein. Bei einem Test mit einer Testbench ist diese die Top-Level-Entity. Die Methode *createPRJFile* erstellt die Projektdatei. Da in dem Test alle benötigten Dateien erstellt und gespeichert werden, sind die Pfade aller benötigten Dateien bekannt und können in der Datei aufgeführt werden. In Listing 5.6 ist eine Projektdatei für ABO gezeigt. Da ABO nicht aus weiteren Teilkomponenten besteht, sind in der Datei nur die VHDL-Datei abo.vhd und die VHDL-Testbench abo_tb.vhd gelistet.

```
1 vhdl work "abo.vhd"
2 vhdl work "abo_tb.vhd"
```

Listing 5.6. Eine Projekt Datei für ABO

Command-Datei

In der Command-Datei werden Einstellungen und Information gespeichert, die für die Simulation nötig sind. Es können Signale definiert werden, die nach der Simulation im *Waveform*-Editor angezeigt werden sollen. Des Weiteren wird in dieser Datei

die Simulationsdauer angegeben. Die Simulationsdauer errechnet sich hier aus der Anzahl der Ticks, die in der ESO Datei definiert sind. Um Fehler durch eine zu kurze Simulationszeit zu vermeiden, wird die Simulationszeitum zwei Ticks erhöht. Die Datei wird mit der Methode *createCMDFile* erstellt. Da in dem Test die ESO Datei bekannt ist, kann die Anzahl der Ticks ohne weiteres gezählt werden. In Listing 5.7 ist eine Command Datei gezeigt, die aus der ESO Datei im Listing 5.2 generiert wurde. Die Simulationszeit für einen Tick beträgt 100ns. In der ESO Datei sind drei Ticks enthalten. Zur Sicherheit werden weitere zwei Ticks hinzuaddiert, was zu einer Simulationszeit von 500ns führt. Eine Command-Datei endet immer mit quit, damit die Simulation auch beendet wird.

```
1 run 500 ns;
2 quit
```

Listing 5.7. Eine Command Datei für ABO

Automatisches Test Skript

Die Skriptdatei wird benötigt, um den Compiler und die Simulation auszuführen und im Anschluss, um die Log-Datei aufzubereiten. In Abbildung 5.8 ist ein automatisch generierte Skriptdatei zum Testen von ABO gezeigt. Das Skript wird mit der Methode *createBatchFile* erzeugt.

```
1  ise_path="/C/Xilinx/14.5/ISE_DS/ISE/"
2  project="abo.prj"
3  toplevelEntity="abo_tb"
4  simulation_tcl="abo.cmd"
5
6  export PLATFORM=nt
7  export XILINX=$ise_path
8  export PATH=$PATH:$XILINX/bin/$PLATFORM
9  export LD_LIBRARY_PATH=$XILINX/lib/$PLATFORM
10
11 binary="tb_abo_isim_beh"
12 compile_params="-intstyle ise -incremental -o "$binary" -prj "$project
13 sim_params="-intstyle ise -tclbatch "$simulation_tcl" -log out.log -sdfnowarn"
14 tmp_out="sim_out.txt"
15
16 fuse $compile_params $toplevelEntity
17 "./"$binary".exe" $sim_params
18 echo -e out.log | cat out.log | grep 'Error:' | sed 's/at.*ps: //' >> $tmp_out
```

Listing 5.8. Eine Batch Datei für ABO

In den ersten vier Zeilen werden Variablen angelegt, die Informationen zum Pfad der ISE Tools, der Projektdatei, Top-Level-Entity und der Command-Datei enthalten. Die folgenden Zeilen fügen den Pfad der ISE Tools zu den Umgebungsvariablen des Betriebssystems hinzu. In der Variable binary steht der Name der Datei, die

Command	Description
-intstyle ise | xflow | silent	Use one of the specified styles for printing messages. Specify ise to format messages for the ISE Console Window or xflow to format messages for XFLOW. Specify silent to suppress all messages.
-incremental	Compiles only the files that have changed since last compile.
-o	Specifies the name of the simulation executable output file.
-prj	Specifies a project file to use for input. A project file contains a list of all the files associated with a design. It is the main source file used by the ISE® software.
-tclbatch <file_name>	Turn batch mode on. By default batch mode is off. When batch mode off, Tcl commands can be issued from the Simulation Console tab even if the engine is busy simulating. When batch mode is on, then all the commands in the specified batch file are executed sequentially until completion, ignoring any commands entered from the command prompt. File_name specifies the name of file containing Tcl commands to be executed.
-log <file_name>	Generates a log file with name specified by file_name.
-sdfnowarn	Do not display SDF warnings.

Tabelle 5.1. Simulations- und Compiler-Optionen von ISE [Inc09]

nach dem Kompilieren die Binärdatei enthält. Die compiler_params und sim_params enthalten spezielle Anweisungen an den Compiler bzw. Simulator. Die Optionen sind in Tabelle 5.1 aufgeführt. Die Log-Datei des Simulators heißt out.log. Die Anweisung fuse startet den Compiler mit den angegebenen Parametern. Die erzeugte Binärdatei wird danach mit den Simulationsparametern gestartet. Die letzte Zeile in der Skriptdatei filtert aus der Log-Datei alle Einträge, in denen ein Error vorkommt. Einträge mit dem Wort Error werden in der Log-Datei von den Assertions aus der Testbench erzeugt. Alle Einträge mit einem Error werden in einer neuen Datei sim_out.txt gespeichert. Diese Datei wird im nächsten Schritt vom JUnit Test analysiert.

Analyse der Log-Datei

In der Log-Datei stehen die gefilterten Ergebnisse der Simulation. Die Datei wird in dem JUnit Test geladen und geöffnet. Wenn in der Datei *Error*-Einträge vorhanden sind, ist der Test fehlgeschlagen, da eine Assertion der Testbench nicht erfüllt wurde. Der JUnit Test löst daraufhin eine JUnit-Exception aus, in der dem Framework mitgeteilt wird, dass der Test fehlgeschlagen ist.

Kapitel 6

Implementierung

Die vorgestellten Transformationen wurden als Teil des KIELER-Projektes entwickelt. Das Projekt basiert auf dem Plugin Konzept von Eclipse. Die verschiedenen Transformationen sind in einzelnen Plugins implementiert. In diesem Kapitel werden Informationen zur Organisation der Plugins und ein kurzer Überblick über die einzelnen Transformationen gegeben. Die detaillierte Beschreibungen der Implementierungsdetails befindet sich in den Kapiteln 4 und 5. Die Implementierung der Transformationen basiert auf den Beschreibungen aus diesen beiden Kapiteln und kann somit nachvollzogen werden. Am Ende des Kapitels werden Details zur Synthese eines Schaltkreises auf einen FPGA vorgestellt. Dazu wird ein kurzer Überblick über Xilinx FPGAs gegeben und eine Implementierung von ABO präsentiert.

6.1 Details zur Eclipse Implementierung

Das KIELER Projekt ist in Plugins organisiert. Tabelle 6.1 listet die vorgestellten Transformationen und die zugehörigen Plugins auf. Die folgenden Kapitel beschreiben Eigenschaften zu jeder Transformationen.

Transformation	**Plugin**
Naive Transformation von SCL zu VHDL, Unterkapitel 4.3	de.cau.cs.kieler.scl.vhdl
Transformation von sequentiellen SCL zu SSA SCL, Unterkapitel 4.4	de.cau.cs.kieler.scl.seqscl.ssascl
Transformation von SSA SCL zu VHDL, Abschnitt 4.4.3	de.cau.cs.kieler.scl.ssascl.vhdl
Transformation von ESO zu Core ESO, Abschnitt 5.1.3	de.cau.cs.kieler.eso.coreeso
Transformation von Core ESO zu einer VHDL Testbench, Abschnitt 5.1.4	de.cau.cs.kieler.eso.vhdl
Automatischer JUnit Test, Unterkapitel 5.3	de.cau.cs.kieler.scl.vhdl.test

Tabelle 6.1. Übersicht der Transformationen und der passenden Eclipse Plugins

6.1.1 Naive VHDL Transformation

Die naive Transformation, welche als erstes in dieser Arbeit vorgestellt wurde, ist im Plugin *de.cau.cs.kieler.scl.vhdl* implementiert. Die Transformation von sequentiellen SCL Code zu VHDL wurde mit Xtend umgesetzt. Die Übersetzung wurde direkt implementiert und ist an die Idee aus dem Unterkapitel 4.3 angelehnt. Die Transformationsmethode erzeugt die benötigte Entity, inklusive der Entity Deklaration und der Architekturbeschreibung. Danach wird ein Prozess angelegt, indem die benötigten Variablen deklariert werden. Jede SCL-Anweisung wird mit dem *expand*-Konstrukt von Xtend in VHDL übersetzt. Das *expand*-Konstrukt stellt eine Art *Function-Overloading* bereit, mit dem die verschiedenen SCL-Anweisungen in den jeweiligen VHDL-Code transformiert werden können.

6.1.2 Sequentielles SCL zu SSA SCL Transformation

In dem Plugin *de.cau.cs.kieler.scl.seqscl.ssascl* ist die Transformation von sequentiellen SCL Code zu einer SCL Repräsentation in SSA-Form implementiert. Diese Transformation wurde ebenfalls mit Xtend umgesetzt. Wenn die Transformation angestoßen wird, wird das SCL Modell, auf das die Transformation angewendet wird, als Parameter übergeben. Zu Beginn wird ein neues SCL Modell angelegt, indem Schritt für Schritt das SSA SCL Modell erzeugt wird. Als erstes werden alle Interface Variablen des Ausgangsmodells in das neue SCL Modell kopiert. Für Ausgabevariablen wird eine pre-Variable angelegt, denn wie in Abschnitt 4.4.2 beschrieben, wird für Ausgaben ein initialer Wert oder der Wert aus dem vorhergehenden Tick benötigt. Nach dem Anlegen der Variablen für das Interface werden die Anweisungen des SCL Codes transformiert. Da es im sequentiellen SCL Code nur Zuweisungen und Conditionals gibt, müssen nur diese beiden Anweisungen unterschieden werden. Die Anweisungen werden entsprechend der im Abschnitt 4.4.2 vorgestellten Idee transformiert und dem neuen SCL Modell hinzugefügt. Ein vollständig transformiertes SSA SCL Modell hat die Dateiendung *.ssa.scl*.

6.1.3 SSA SCL zu VHDL Transformation

Die Transformation von SSA SCL Modellen zu VHDL ist in dem Plugin *de.cau.cs.kieler.ssascl.vhdl* implementiert. Die Xtend Transformation nimmt als Eingabe ein SCL-Modell, welches in der SSA-Form vorliegen muss. Zu Beginn der Transformation wird die Interface Deklaration des SCL-Modells angepasst. Das bedeutet, es werden alle Ein-Ausgabevariablen in einzelne Ein- und Ausgabevariablen aufgetrennt, wie dies bereits im Abschnitt 4.4.3 beschrieben wurde. Danach wird die Entity Deklaration und die Architekturbeschreibung angelegt. In dieser Transformation werden die SCL-Anweisungen in parallele VHDL-Anweisungen übersetzt und nicht in Prozesse wie in der naiven Lösung. Normale Zuweisungen werden direkt unter der Verwendung der VHDL-Syntax übersetzt. Conditionals werden in bedingte Entscheidungen in VHDL

transformiert. Zum Schluss werden die Sampling Register und die Pre-Register mit Hilfe von Prozessen angelegt. Ebenso werden die Register für das Reset-Signal und die GO-Signal Generierung erzeugt, siehe Abschnitt 4.4.4. Die transformierte VHDL-Datei hat die Dateiendung *.vhd*.

6.1.4 ESO zu Core ESO Transformation

Damit die in Abschnitt 5.1.1 beschriebenen ESO Dateien für das Testen von SCCharts verwendet werden können, müssen sie in Core ESO Dateien übersetzt werden. Diese M2M-Transformation ist in dem Plugin *de.cau.cs.kieler.eso.coreeso* implementiert. Die Transformation nimmt ein ESO Modell und transformiert alle Signale aus jedem Trace und jedem Tick in Variablen. Diese Transformation wird im Abschnitt 5.1.3 ausführlich beschrieben. Core ESO Dateien haben die Endung *.core.eso*.

6.1.5 Core ESO zu VHDL Testbench Transformation

Das Erzeugen einer Testbench aus einer Core ESO Datei geschieht im Plugin *de.cau.cs.kieler.eso.vhdl*. Diese Codegenerierung übersetzt eine gegebene Core ESO Datei in eine VHDL-Testbench. Die Funktion der Transformation wird in Abschnitt 5.1.4 detailliert erklärt. Diese Transformation wurde ebenfalls mit Xtend programmiert und Testbench Dateien haben die Dateiendung *.vhd*. Damit die Testbench und das transformierte Modell auseinandergehalten gehalten werden können, hat die Testbench den Namenszusatz *_tb* bekommen.

6.1.6 Automatischer JUnit Test

Der automatische Test ist im Plugin *de.cau.cs.kieler.scl.vhdl.test* implementiert. Details zu dieser Implementierung sind im Unterkapitel 5.3 zu finden. Der Test wurde in Java geschrieben und es wurde das Junit Framework verwendet. Damit der Test und insbesondere die automatische Ausführung der Xilinx Werkzeuge verwendet werden können, müssen der Pfad der Xilinx Tools (*ISE_PATH*) und der Pfad zu den Testmodellen (*MODELS_REPOSITORY_PATH*) in der Java-Klasse angegeben werden. Dazu befinden sich am Anfang des Codes der Transformation entsprechende Felder, siehe Listing 6.1. Der Test kann aus einer Eclipse Instanz gestartet und die Ergebnisse überprüft werden.

6.2 Hardwaresynthese

Bei der Hardwaresynthese ist es das Ziel, abstrakt beschriebene Modelle in Hardware abzubilden. Die abstrakte Beschreibung von SCCharts wurde in mehreren Transformationsschritten zu VHDL übersetzt. Die VHDL Repräsentation kann in einen

```
1  public class SCLVHDLAutomatedJUnitTest {
2
3      // The Constant where to find the models and ESO files
4       static final IPath MODELS_REPOSITORY_PATH = new Path("../../../models/SCCharts/");
5
6      // Path from ISE distribution, used for systems PATH variable
7       private static String ISE_PATH = "/C/Xilinx/14.5/ISE_DS/ISE/";
8
9      //Test Method
10     public void SCLVHDLTestRunnerExecution() throws IOException, InterruptedException {
11         // Performs the JUnit Test for every Model
12         // – from SCCharts to VHDL
13         // – from ESO to VHDL Testbench
14         // – run Xilinx compiler
15         // – test Results
16     }
17 }
```

Listing 6.1. Codeausschnitt der JUnit Testklasse — Konstanten Definition

äquivalenten Schaltkreis abgebildet werden. Solche Schaltkreise wurden zum Beispiel in Abbildung 4.39 vorgestellt. Dieser Schaltkreis wird, so wie er vorliegt, jedoch nicht auf einem FPGA realisiert. Um zu verdeutlichen, wie die Funktion des Schaltkreises auf einen FPGA gebracht werden, ist im Folgenden der Aufbau eines FPGAs erläutert und die Idee des *Place and Route*-Algorithmus vorgestellt.

6.2.1 Aufbau eines FPGAs

Ein FPGA ist ein hochintegrierter digitaler Schaltkreis in der Digitaltechnik. Auf FPGAs werden im Gegensatz zu herkömmlichen Microcontrollern Schaltkreise beschrieben. Es werden auf einem FPGA somit Strukturen und keine ablaufenden Programme ausgeführt. Die größten FPGA Hersteller sind Xilinx und Altera[1]. Sie stellen eine große Produktpalette verschiedener FPGAs, für die unterschiedlichsten Anwendungsfälle, zur Verfügung. FPGAs sind nach einem ähnlichen Schema aufgebaut. Im Abbildung 6.1 ist der schematische Aufbau eines Xilinx FPGAs zu sehen. Ein FPGA basiert dabei auf einer Matrix von logischen Zellen, den Configurable Logic Blocks (CLBs) und programmierbaren Verbindungen sowie ein Netz an vorhandenen Leitungen (Verdrahtungskanälen). In der Abbildung sind weiterhin Ein-Ausgabe-Ports (*IOB*) und RAM-Speicher (BRAM) zu sehen. Benötigte Taktsignale könne mit Hilfe des Digital Clock Managers (DCMs) erzeugt werden.

Configurable Logic Block

Das Verhalten, das ein synthetisierter Schaltkreis auf einem FGPA widerspiegeln soll, wird mit den CLBs abgebildet. Ein CLB ist die logische Einheit eines FPGAs.

[1] http://www.altera.com/

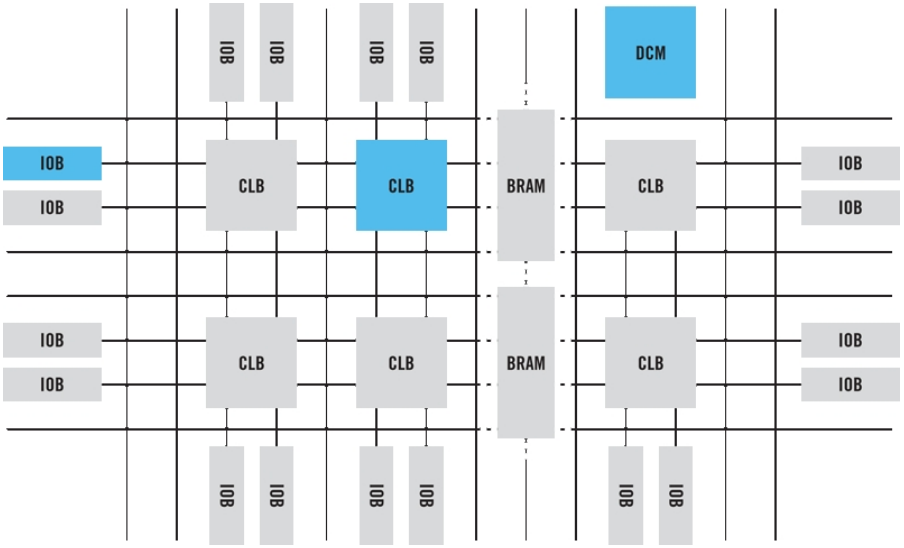


Abbildung 6.1. Block Struktur eines FPGAs [Inc13]

Die genaue Anzahl und die Eigenschaften variieren zwischen den verschiedenen FPGAs. Um das Verhalten einer Schaltung abzubilden, werden die CLBs entsprechend konfiguriert und untereinander verbunden. Zwei CLBs sind in Abbildung 6.2 (links) abgebildet. Ein CLB besteht in der Regel aus zwei *Slices* (zum Beispiel: Slice X0Y0 und X0Y1 in der Abbildung) und einer Switch Matrix, die mit einem weiteren CLB geteilt wird [Inc12c]. Die Switch Matrix beinhaltet konfigurierbare Verbindungen, mit denen unter anderem einzelne Verbindungen zwischen den Slices hergestellt werden können. Die genaue Funktion einer Sclice kann zwischen den verschiedene FPGAs variieren, jede Slice enthält jedoch mindestens eine konfigurierbare Schaltmatrix mit vier oder sechs Eingängen, einige Multiplexer (MUXF5) und Register (Register/Latch,CY), siehe Abbildung 6.2. Die Schaltmatrix ist äußerst flexibel und kann zum Beispiel als kombinatorische Logik (LUT F), als *Shift-Register* (SRL16) oder als *RAM* (RAM16) konfiguriert werden. Die beiden Slices einer CLB sind nicht direkt miteinander verbunden, sie können jedoch über Switch Matrix miteinander verbunden werden.

6.2.2 Place and Route Algorithmus

Der *Place and Route*-Algorithmus berechnet, wie der synthetisierte Schaltkreis auf dem jeweiligen FPGA konfiguriert werden kann. Dazu wird das Verhalten des Schaltkreises in entsprechenden CLBs abgebildet und auf dem Chip platziert. Anschließend werden die benötigten CLBs mit Hilfe des *Route*-Algorithmus miteinander verbunden. Mit den programmierbaren Verbindungen (Switch Matrix) und den Verdrahtungskanälen lassen sich beliebige Verbindungen zwischen den einzelnen CLBs herstellen.

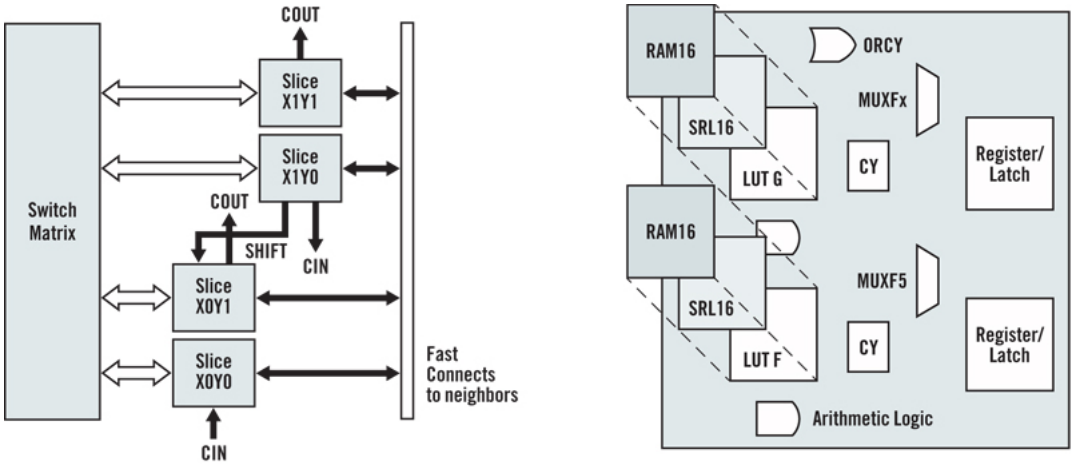


Abbildung 6.2. Struktur zweier CLBs (links) und einer Slice (rechts) [Inc13]

Der Algorithmus muss bei der Berechnung darauf achten, welche Flächen des Chips bereits belegt sind und dass alle benötigten Verbindungen, auch zu Ein-Ausgabeports, hergestellt werden können. Ebenso können Anforderungen an die maximale Taktrate der Schaltung gestellt werden, die ebenfalls beim Platzieren und Routen berücksichtigt werden müssen. In Abbildung 6.3 ist ein Ausschnitt der internen Sicht eines FPGAs zu sehen, auf dem ABO plaziert und geroutet wurde. Die beiden blauen Kästen sind CLBs und die rot hervorgehobenen Leitungen, sind die Leitungen, die geroutet wurden um die CLBs zu verbinden. Die grünen Leitungen hingegen werden nicht verwendet.

Die interne Sicht einer Slice ist in Abbildung 6.4 dargestellt. Die cyan eingefärbten Leitungen werden benötigt, um die entsprechende Funktion abzubilden. Die magenta gefärbten Leitungen hingegeben sind Leitungen die in der Sclice vorhanden sind, jedoch nicht verwendet werden. Die beschriebenen Komponenten aus Abschnitt 6.2.1 sind zum Teil wiederzuerkennen, zum Beispiel: Register, Multiplexer und die Schaltmatrix (großer Baustein links). In dem unteren Schaltungsteil der abgebildeten Slice, werden Signalwerte für zwei pause Anweisungen in ABO berechnet.

Im nächsten Kapitel wird ein vollständige Implementierung von ABO auf realer Hardware vorgestellt.

6.2.3 Implementierung von ABO auf einem FPGA

Die Simulationen der synthetisierten Schaltkreise konnte die korrekte Funktion der Transformation belegen. Um die simulierten Ergebnisse unter realen Testbedingungen zu bestätigen, wurde ABO auf ein FPGA synthetisiert. Am Lehrstuhl stand ein Evaluationsboard von Xilinx (ML605) zur Verfügung, siehe Abbildung 6.5. Das Board ist mit einem Xilinx Virtex 6 bestückt [Inc12d].

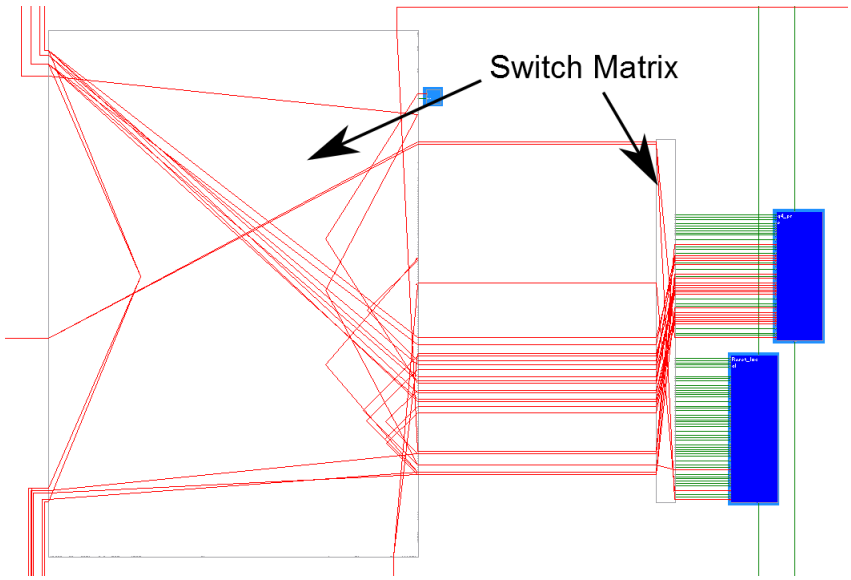


Abbildung 6.3. Sicht auf eine FPGA mit zwei gerouteten Slices

Das Board stellt eine große Anzahl verschiedener Möglichkeiten bereit, Programme für die unterschiedlichsten Aufgaben zu entwickeln und zu testen. So stehen dem Programmierer zum Beispiel ein DVI-Ausgang, eine Netzwerk- und USB-Schnittstelle, ein DDR3 Speicher und einfache Taster sowie LEDs zur Verfügung. Das FPGA wird über ein USB-Kabel mit dem Rechner verbunden, mit dem er programmiert werden soll.

In Abbildung 6.6 ist ein Blockschaltbild gezeigt, das den prinzipiellen Testaufbau von ABO darstellt. Das ABO-Beispiel eignet sich sehr gut für einen Test, denn ABO besitzt nur boolsche Datentypen. Diese können gut durch Taster für Eingaben und LEDs für Ausgaben umgesetzt werden. ABO wurde unter der Verwendung der Drucktaster und der LEDs auf den FPGA programmiert (siehe Abbildung 6.5 Push Buttons). Die Taster werden für die Eingaben A, B, Reset und Tick verwendet. Die LEDs signalisieren die Werte der Ausgaben A_out, B_out, O1 und O2.

Mechanische Taster haben die Eigenschaft, bei Betätigung zu prellen. Das bedeutet, es werden mehrere Tastvorgänge nacheinander ausgeführt. Damit die Schaltung des FPGA pro Tastendruck nur einen Impuls erhält, wurden Entprelleinheiten (Debouncer) hinzugefügt. Um eine möglichst große Flexibilität beim Testen zu behalten, wurde der Tick ebenfalls durch einen Taster realisiert. Die Verwendung eines automatisch generierten Taktes erschwert das Testen, da man die Eingaben in Abhängigkeit des Signals tätigen muss.

Das ABO Beispiel wurde mit den vorgestellten Transformationen in eine VHDL Datei transformiert. Die Entprelleinheiten wurden anschließend manuell hinzugefügt. Um eine Verbindung der internen Signale zur Außenwelt, den Tastern und LEDs zu

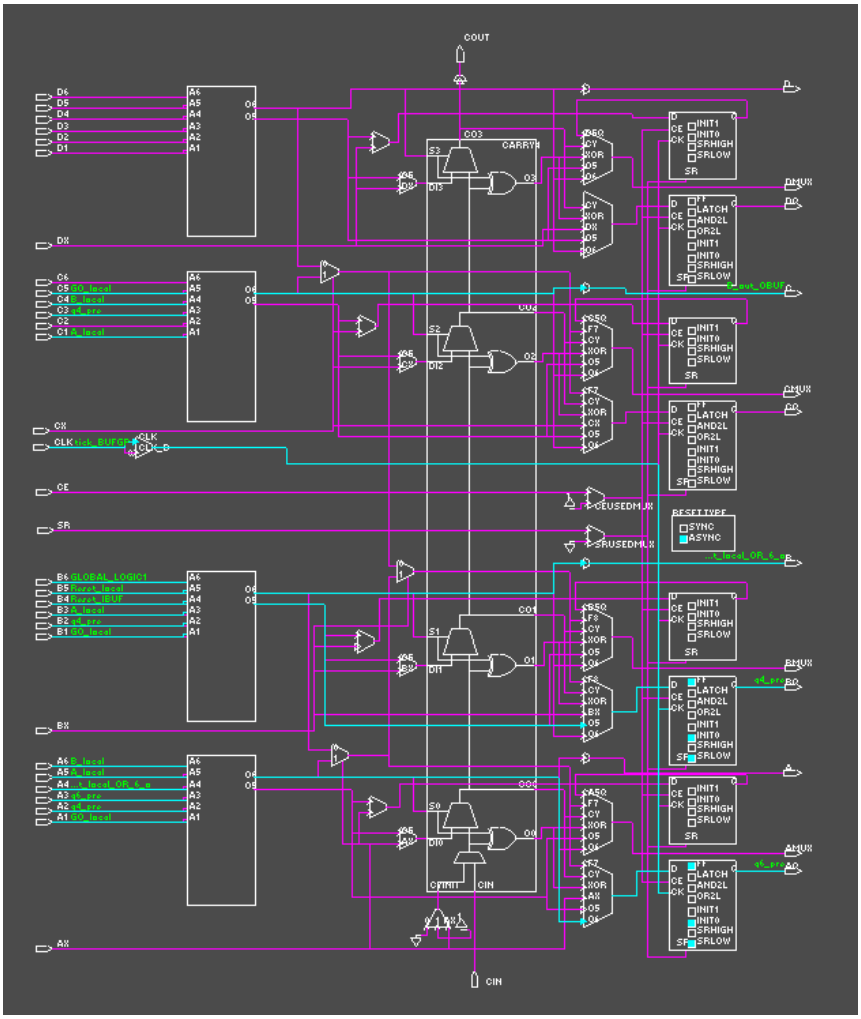


Abbildung 6.4. Die interne Sicht einer konfigurierten Slice

bekommen, wird in einer UCF-Datei (User Constraint File) die genaue Zuordnung zwischen internen Signalen und den FPGA-Anschlusspins festgelegt. ISE synthetisiert den Schaltkreis für den ausgewählten FPGA und das Xilinx Impact Tool überträgt die Schaltung auf den FPGA.

Die Funktion von ABO befindet sich nun auf dem FPGA und kann mittels der Taster und LEDs überprüft werden. Der manuelle Test konnte das korrekte Verhalten von ABO bestätigen. Im nächsten Kapitel werden zwei Ausführungen auf der Hardware gezeigt.

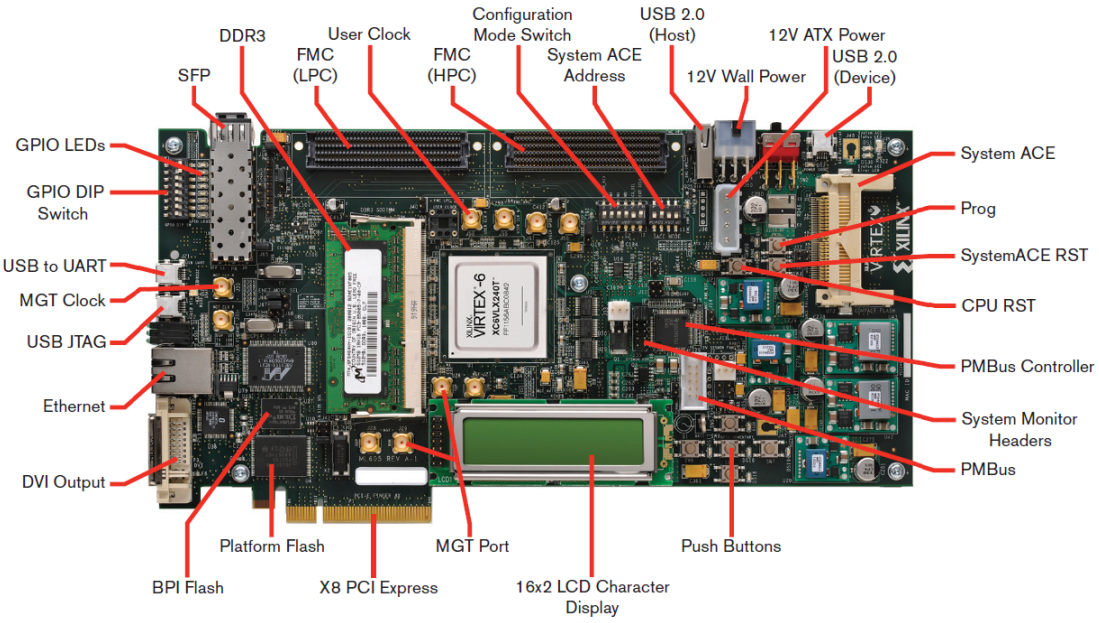


Abbildung 6.5. Xilinx Entwicklungsplatine — ML605 [Inc12d]

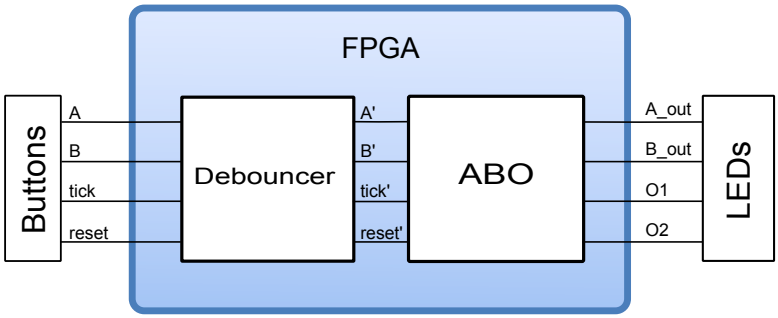


Abbildung 6.6. Übersicht der ABO Hardware Implementierung

ABO auf Hardware

In diesem Kapitel werden zwei Ausführungspfade für ABO auf dem Evaluationsborad ausgeführt. Dazu wurden, wie im Kapitel zuvor beschrieben, die Taster für Eingaben und die LEDs zur Signalisierung der Ausgaben verwendet. In Abbildung 6.7 sind die beiden Pfade und die dazugehörigen Reaktionen in Form von Bildern gezeigt.

Die Belegung der Taster und LEDs sind in den jeweiligen Abbildungen aufgeführt. Die Taster sind in Himmelsrichtungen angeordnet. Der Nord-Taster ist mit dem internen Reset verbunden. Der östliche Taster dient zur Simulation des Ticks. Das Signal B kann mit dem Taster im Süden und das Signal A mit dem Taster im Westen erzeugt werden. Die LEDs sind neben den Tastern angeordnet und sind wie folgt belegt: Nord: O1, Osten: O2, Süden: B_out und Westen A_out.

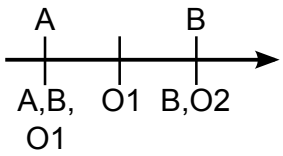
Damit die Schaltung Eingaben verarbeiten kann, muss zuerst der Taster, der das gewünschte Signal erzeugt, gedrückt werden, und während der Taster gedrückt wird, muss der Tick erzeugt werden. Nachdem die Berechnung gestartet ist, können die Taster losgelassen werden. Die LEDs zeigen die aktuell berechneten Werte an. Die Werte werden solange angezeigt, bis sie durch eine neue Berechnung überschrieben werden.

Zu dem ersten Ausführungspfad (Abbildung 6.7a) gehören die Bilder 6.7b, 6.7c und 6.7d. Jedes Bild stellt einen Tick dar. So wurde in Abbildung 6.7b der Taster A gedrückt und als Ausgaben wurden A_out, B_out, O1 berechnet. Die Reaktion entspricht genau dem ersten Tick. Im zweiten Tick wird keine Eingabe gesetzt, sondern nur ein Tick ausgeführt. Es leuchtet daraufhin nur die LED für O1. Im letzten Tick, im ersten Pfad, wird B gesetzt und ein neuer Tick ausgeführt. Als Ausgabe erscheinen B_out und O2. Die Ausführung ist danach beendet.

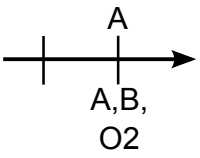
Durch Drücken des Reset-Tasters und Generieren eines Ticks (Tick-Taster), wird der Schaltkreis zurückgesetzt, sodass ein neuer Ausführungspfad berechnet werden kann. Es muss ein Tick generiert werden, damit die Schaltung zurückgesetzt wird, da es sich in der Schaltung um Register mit einem taktsynchronen Reset handelt und die in Abschnitt 4.4.4 vorgestellte zweite Reset Variante implementiert ist.

Der zweite Ausführungspfad, der getestet wurde, ist in Abbildung 6.7e zu sehen. Die entsprechenden Reaktionen sind in den Bildern 6.7f und 6.7g gezeigt. Im ersten Tick werden keine Eingaben gesetzt. Es wird nur ein Tick durch Drücken des Tick-Tasters erzeugt. In diesem Tick existieren keine Ausgaben. Im zweiten Tick wird A gesetzt und die Ausgaben A_out, B_out, O2 berechnet, siehe Abbildung 6.7g.

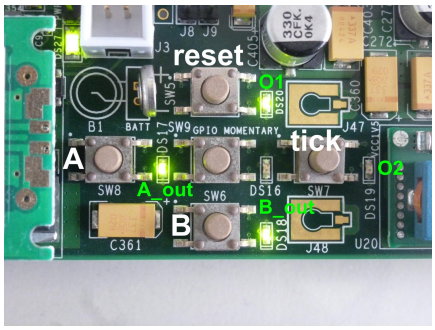
Der synthetisierte Schaltkreis hat das erwartete Verhalten bestätigt. Es konnten beide Ausführungspfade auf realer Hardware nachgestellt werden. Der Schaltkreis hat zu den Eingaben die erwarteten Ausgaben produziert.



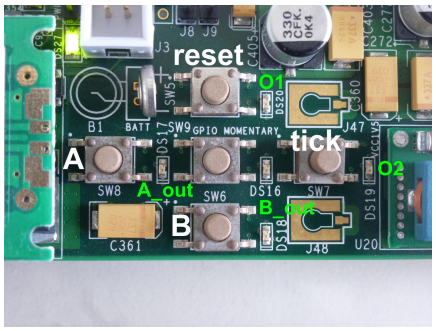
(a) ABO auf Hardware — Ausführungspfad A



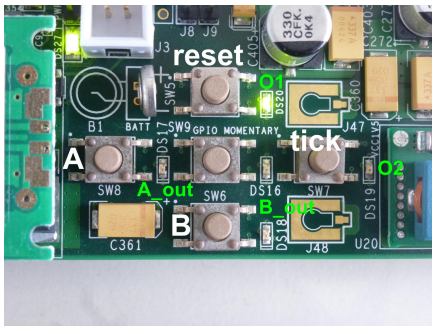
(e) ABO auf Hardware — Ausführungspfad B



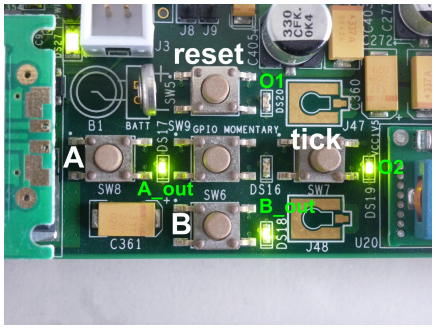
(b) ABO auf Hardware — Ausführungspfad A — erster Tick



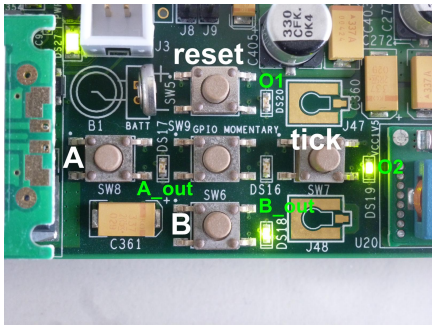
(f) ABO auf Hardware — Ausführungspfad B — erster Tick



(c) ABO auf Hardware — Ausführungspfad A — zweiter Tick



(g) ABO auf Hardware — Ausführungspfad B — zweiter Tick



(d) ABO auf Hardware — Ausführungspfad A — dritter Tick

Abbildung 6.7. ABO auf Hardware — zwei Ausführungspfade

Kapitel 7

Evaluierung

In diesem Kapitel werden die naive und die SSA-basierende Transformationen aus den Unterkapiteln 4.3 und 4.4 evaluiert. Die Evaluierung soll herausstellen, welche Chipauslastung ein transformierter Schaltkreis aufweist und mit welcher minimalen Periodendauer des Tick-Signals der Schaltkreis ausgeführt werden kann.

Es werden zwei Evaluierungen durchgeführt. Die erste Evaluierung verwendet flache SCCharts, ohne Hierarchie und nebenläufige Regionen, mit einer steigenden Anzahl an Zuständen. In der zweiten Evaluation werden hierarchische SCCharts verwendet, wobei die Anzahl der Hierarchieebenen erhöht wird. Es werden für jedes SCChart jeweils die Chipauslastung und die Periodendauer, mit der der Schaltkreis ausgeführt werden kann, verglichen.

7.1 Testverfahren

Die verwendeten SCCharts wurden automatisch generiert und mit der einfachen und der SSA-basierten Transformation in VHDL-Dateien übersetzt. Für jede der erzeugten VHDL-Dateien wurde ein Projekt in der Entwicklungsumgebung ISE angelegt. In einem ersten Schritt wird die VHDL-Datei mit ISE synthetisiert. Das bedeutet, dass die VHDL-Beschreibung in einen Schaltkreis übersetzt wird. Anschließend wurde der Schaltkreis von ISE in mehreren Schritten für die Implementierung auf einem FPGA übersetzt. Die Implementierungsroutine von ISE erstellt eine entsprechende Repräsentation, in der der Schaltkreis auf dem entsprechenden FPGA platziert und geroutet wird (Place and Route Algorithmus, siehe Abschnitt 6.2.2). Die Synthese- und Implementierungsroutinen erstellen detaillierte Zusammenfassungen der einzelnen Vorgänge. Es werden zum Beispiel Informationen zur Anzahl der verwendeten Slices, der verwendeten Register und Look-Up-Tabellen, aber auch zur minimalen ermittelten Periodendauer aufgeführt. Für die Evaluierung werden die verwendeten Slices und die ermittelte Periodendauer verwendet.

Um die beiden Transformationen miteinander vergleichen zu können, wurden die selben Einstellungen für alle transformierten SCCharts in ISE verwendet. Im ISE wurden die Standardeinstellungen verwendet und keine weiteren Optimierungen eingestellt. Da sich die einzelnen FPGAs in ihrem Funktionsumfang und dem internen Aufbau unterscheiden, wurde für die Messungen einheitlich ein Virtex 6 (XC6VLX75t) FPGA verwendet.

Die Chipauslastung wurde anhand der belegten Slices des FPGAs gemessen und konnte der Zusammenfassung der Implementierungsroutine entnommen werden. Gleiches gilt für die minimale Periodendauer für einen Schaltkreis, diese kann ebenfalls der Zusammenfassung entnommen werden. Die Werte wurden für jedes generierte Modell ermittelt und ausgewertet. Die ermittelten Werte können nicht als allgemeingültige Werte angesehen werden, es müssen einige Besonderheiten berücksichtigt werden, die im folgenden Kapitel erläutert werden.

7.2 Besonderheiten der Evaluierung

Eine allgemeingültige Evaluierung der synthetisierten Schaltkreise ist nicht möglich. Die Chipauslastung und die minimale Periodendauer eines synthetisierten Schaltkreises, sind wesentlich vom verwendeten FPGA und von den Einstellungen des Place and Route Algorithmus abhängig.

Das Ergebnis des Place and Route Algorithmus hängt von verschiedenen Faktoren ab. Zum Einen ist das Ergebnis vom verwendeten FPGA und dem internen Aufbau abhängig. Der Aufbau der CLBs variiert zwischen verschieden FPGAs [Inc13]. So kann der Algorithmus eine Schaltung bei einem FPGA in zwei Slices unterbringen, während er bei einem anderen FPGA zum Beispiel vier Slices benötigt. Des Weiteren können vor der Implementierung Angaben gemacht werden, mit welcher Periodendauer der vorliegende Schaltkreis später ausgeführt werden soll. Der Algorithmus berechnet entsprechend der gewünschten Periodendauer die Platzierung des Schaltkreises auf dem FPGA, damit er mit der vorgegebenen Periodendauer ausgeführt werden kann. Durch diese Vorgabe können die Platzierung, die verwendeten Slices und die minimale Periodendauer indirekt beeinflusst werden. Ebenso beeinflussen schon belegte Chipflächen das Routing und somit auch die Anzahl der verwendeten Slices und die erreichbare Periodendauer, denn im schlechtesten Fall könnte der Schaltkreis über den gesamten Chip verteilt werden. Die entstehenden Leitungslängen wirken sich negativ auf die minimale Periodendauer aus. Damit diese Eigenschaften bei allen Tests einheitlich sind, wurde für die Evaluierung ein einheitlicher FPGA ohne weitere belegte Chipfläche verwendet. Für jeden Vergleich der beiden erzeugten Schaltkreise, aus den jeweiligen Transformationen, wurde die selbe zu erzielende Periodendauer verwendet.

Aus den genannten Gründen kann keine allgemeine Aussage darüber getroffen werden, welche Anzahl an Slices ein SCChart mit einer bestimmten Anzahl an Zuständen benötigt oder mit welcher minimalen Periodendauer es ausgeführt werden kann. In den folgenden beiden Abschnitten werden die Evaluationsergebnisse vorgestellt.

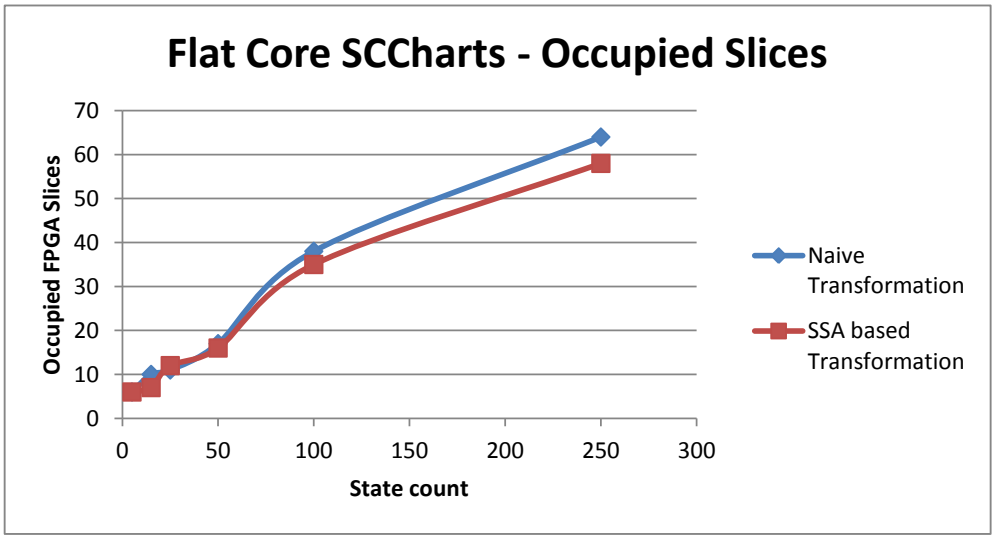


Abbildung 7.1. Evaluationsergebnis flacher Core SCCharts — belegte Slices

7.3 Evaluierung von Core SCCharts ohne Hierarchie und nebenläufige Regionen

In dem ersten Vergleich der Transformationen werden automatisch generierte flache Core SCCharts verwendet. Das bedeutet, dass die Core SCCharts keine nebenläufigen Regionen und keine Hierarchien besitzen. Um die Chipauslastung und die minimale Periodendauer der erzeugten Schaltkreise vergleichen zu können, wurden Core SCCharts mit einer unterschiedlichen Anzahl an Zuständen verwendet. Jeder Zustand besitzt zwei ausgehende Transitionen.

In Abbildung 7.1 ist die Chipauslastung in Form von verwendeten Slices eines FPGAs, bezogen auf die Anzahl der Zustände, die das Core SCChart hat, dargestellt. Das Verhältnis zwischen den belegten Slices und der Anzahl der Zustände lässt auf einen annähernd linearen Verlauf schließen. Die Unregelmäßigkeiten der Messungen sind vermutlich auf den Place and Route Algorithmus zurückzuführen. Weiterhin werden für die Schaltkreise, die mit der SSA-basierten Transformation übersetzt wurden, überwiegend weniger Slices benötigt, als für Schaltkreise, die mit der ersten einfachen Transformation übersetzt wurden. Dies schlägt sich jedoch erst bei Core SCCharts mit einer größeren Anzahl an Zuständen nieder. Da der Unterschied jedoch sehr gering ist, müssten zusätzliche Messungen mit SCCharts mit weitaus mehr Zuständen klären, ob sich die These bestätigt oder ob dieser geringe Unterschied auf den Place and Route Algorithmus zurückführen lässt.

In der zweiten Messung sollen die Ausführungsgeschwindigkeiten der Schaltkreise, die durch die verschiedenen Transformationen erzeugt wurden, verglichen werden. Für die Messungen wurde die Vorgabe der minimalen Periodendauer für den

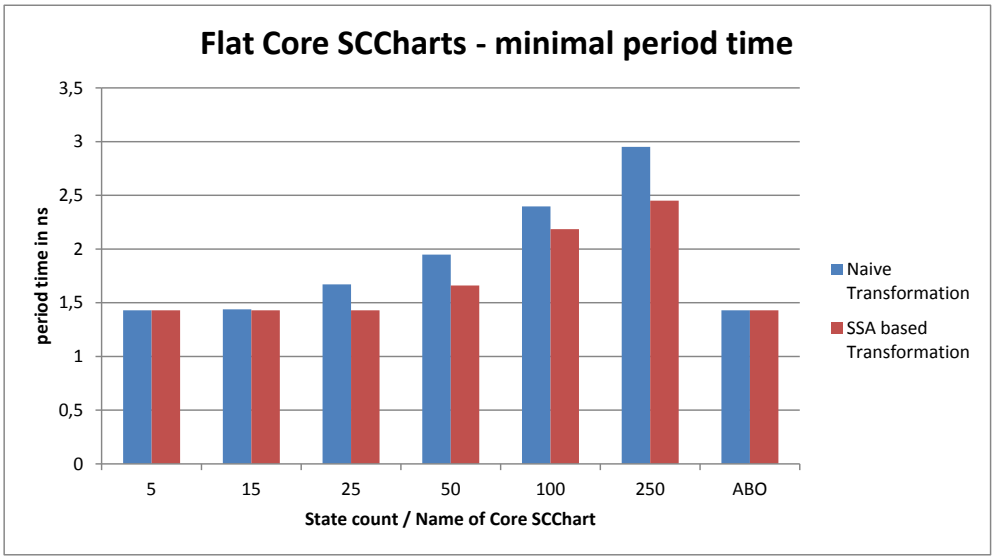


Abbildung 7.2. Evaluationsergebnis flacher Core SCCharts — minimale Periodendauer

Place and Route Algorithmus schrittweise verringert, bis der Algorithmus nicht mehr in der Lage war, die Anforderungen zu erfüllen. Diese Messung zeigt nur eine Tendenz und darf nicht als absoluter Messwert angesehen werden. Sie soll lediglich zeigen, ob die synthetisierten Schaltkreise der beiden Transformationen, mit annähernd gleicher Geschwindigkeit ausgeführt werden können oder ob eine Transformation einen deutlich schnelleren Schaltkreis erzeugt hat.

In Abbildung 7.2 werden die verschiedenen Ausführungsgeschwindigkeiten der synthetisierten Core SCCharts gezeigt. Ebenfalls ist das ABO Beispiel aufgeführt. Bei kleineren SCCharts (bis 15 Zustände) ist die Periodendauer annähernd gleich. Bei größeren SCCharts hingegen sind die Schaltkreise, die mit der SSA-basierenden Transformation übersetzt wurden, schneller.

7.4 Evaluierung hierarchischer Core SCCharts

Im zweiten Teil der Evaluation werden für hierarchische Core SCCharts die Chipauslastung und die Periodendauer bestimmt. Für diese Messung wurden fünf, ebenfalls automatisch generierte, Core SCCharts verwendet, deren Hierarchietiefe mit jeder Messung vergrößert wurde. Dabei enthält jeder Makrozustand zwei parallele Regionen mit jeweils fünf Zuständen. Jede Hierarchiestufe enthält wiederum zwei Regionen und fünf Zustände. Wie in der ersten Evaluation besitzt jeder Zustand zwei ausgehende Transitionen.

In der Abbildung 7.3 ist die Anzahl der belegten Slices, bezogen auf die Hierarchieebenen dargestellt. Aufgrund der quadratisch steigenden Anzahl der Zustände,

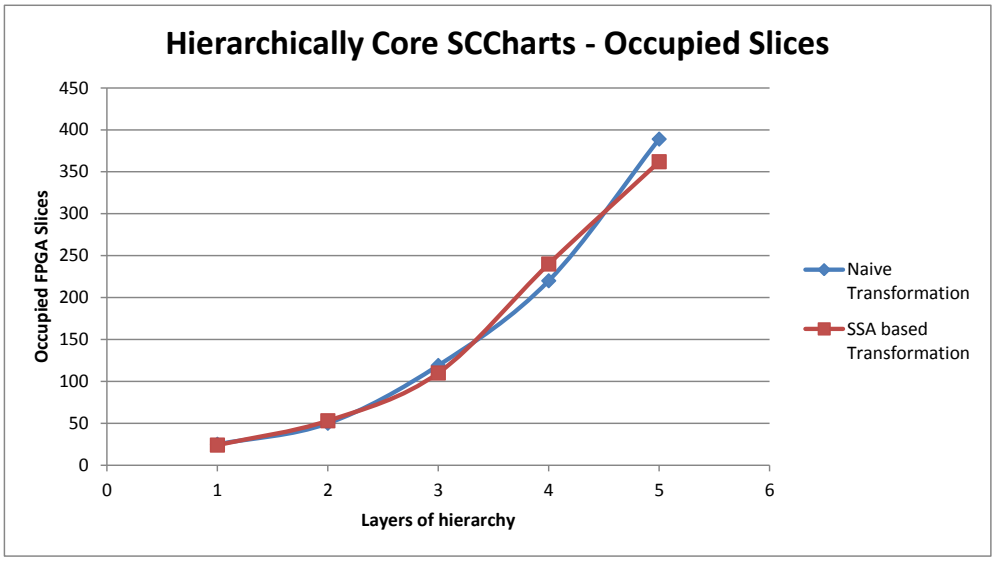


Abbildung 7.3. Evaluationsergebnis hierarchischer Core SCCharts — belegte Slices

durch das Hinzufügen von Hierarchieebenen, steigt auch die Anzahl der verwendeten Slices quadratisch. Die Anzahl der verwendeten Slices ist somit nicht von der Hierarchietiefe abhängig, sondern von der Anzahl der Zustände. Die Anzahl der verwendeten Slices verhält sich in hierarchischen Modellen weiterhin annähernd linear zur Anzahl der Zustände. Die unterschiedliche Anzahl der Slices zwischen den beiden transformierten Schaltkreisen, ist auch in diesem Fall nicht eindeutig zu bewerten. So benötigt das transformierte SCCharts mit vier Hierarchieebenen nach der SSA-basierten Transformation mehr Slices als nach der naiven Transformation. Bei dem transformierten SCCharts mit fünf Hierarchieebenen benötigt der SSA-basierte Schaltkreis hingegen weniger Slices als nach der naiven Transformation. Die Unterschiede sind für die Anzahl der Zuständen gering, so dass auch in dieser Messung größere SCCharts einen signifikanten Unterschied herausstellen müssten.

Es wurde ebenfalls die minimale Periodendauer hierarchischer Modelle validiert. Die Messwerte wurden nach dem selben Verfahren wie in der Messung flacher SCCharts ermittelt. Die Abbildung 7.4 zeigt die Auswertung der Messungen. Die Ausführungsgeschwindigkeiten unterschieden sich zwischen den beiden Transformationen nur minimal. Man würde erwarten, dass der Unterschied der Periodendauer der beiden Transformationen mit der steigenden Anzahl der Zustände größer wird, wie bei der Messung der flachen SCChart. Das dies nicht Fall ist, kann damit Begründet werden, dass bei den getesteten hierarchischen SCCharts die nebenläufigen Regionen relativ kleine Zustandsfolgen beinhalten. Die Periodendauer ist vom längsten Pfad der generierten Logik abhängig. Diese wiederum ist vom längsten Pfad des SCCharts abhängig. Da bei den hierarchischen SCCharts die Anzahl der nebenläufi-

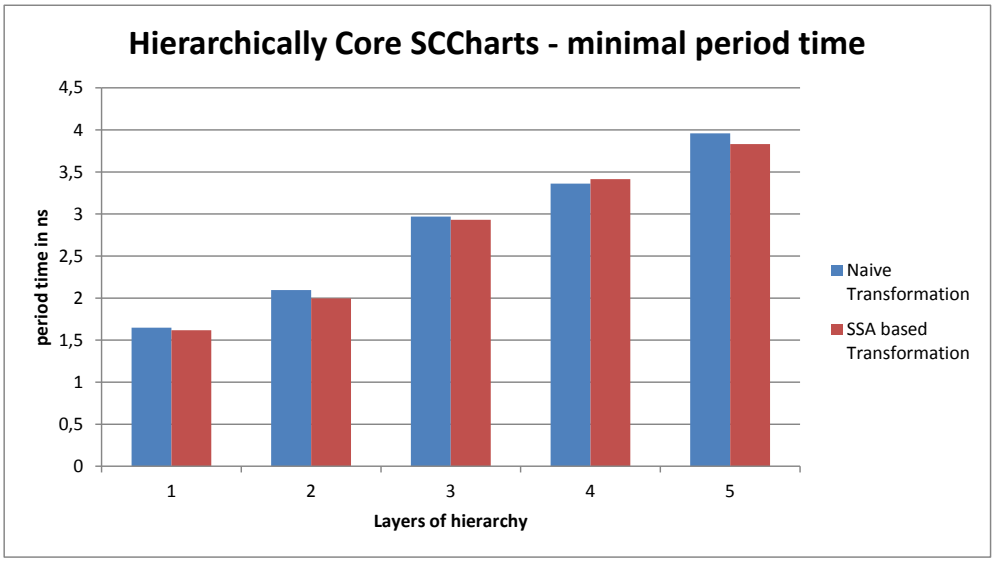


Abbildung 7.4. Evaluationsergebnis hierarchischer Core SCCharts — minimale Periodendauer

gen Regionen erhöht wird und nicht die Länge der Zustandsfolgen, unterscheiden sich die Ausführungsgeschwindigkeiten, im Gegensatz zu flachen SCCharts, kaum. Messungen mit weit aus größeren SCCharts bezüglich der Anzahl der Regionen und der Länge der internen Zustandsfolgen, könnten einen weiteren Aufschluss darüber geben, ob sich die Ausführungsgeschwindigkeiten der beiden Transformation bei diesen größeren SCCharts unterscheiden.

7.5 Fazit der Evaluation

Als abschließende Bewertung der Evaluation kann festgehalten werden, dass die Slices, die benötigt werden, um ein synthetisiertes Core SCCharts auf einem FPGA abzubilden, sich annähernd linear zur Anzahl der verwendeten Zustände verhält. Dies gilt für flache sowohl als auch für hierarchische SCCharts. Jedoch können flache und hierarchische SCCharts nicht direkt miteinander verglichen werden. Hierarchische SCCharts belegen gegenüber flachen SCCharts bei gleicher Zustandsmenge mehr Slices. Das kann damit begründet werden, dass hierarchische SCCharts mit nebenläufigen Abhängigkeiten einen größeren Scheduling-Aufwand erfahren, der sich in einer ausgeprägteren Guard Berechnung widerspiegelt. Diese aufwändigere Berechnung wirkt sich direkt auf die Anzahl der Slices aus.

Ebenso lässt sich eine Tendenz vermuten, dass mit größeren SCCharts die Anzahl der benötigten Slices bei der SSA-basierenden Transformation geringer ist, als bei der naiven Transformation.

Bei der minimalen Periodendauer hingegen wurde bei flachen Core SCCharts bis zu 250 Zuständen festgestellt, dass die SSA-basierte Transformation tendenziell schnellere Schaltkreise erzeugt, je mehr Zustände das Core SCChart hat. Bei hierarchischen Modellen hingegen ist kaum ein Unterschied festzustellen. Das kann damit begründet werden, dass die längsten Pfade innerhalb des hierarchischen SCCharts noch zu kurz sind, um wie bei der Messung der flachen SCCharts eine Differenz der Periodendauer zwischen den beiden Transforationen deutlich zu machen. Denn die minimale Periodendauer wird durch den längsten Pfad des SCCharts bestimmt.

Kapitel 8

Fazit

Nachdem die einzelnen Transformationsschritte und deren Implementierung sowie eine Idee zum Testen derartiger Transformationen gegeben wurde, wird im letzten Kapitel dieser Arbeit eine Zusammenfassung der Arbeit präsentiert. Die Zusammenfassung schließt mit weiteren Ideen zur Verbesserung und Weiterentwicklung der Transformationen, die noch umgesetzt werden können, ab.

8.1 Zusammenfassung

In dieser Arbeit wurde eine Lösung präsentiert, wie aus SCCharts Hardware synthetisiert werden kann. Dazu wurden zu Beginn zwei Transformationen von Smyth [Smy13] vorgestellt, die in den hier vorgestellten Ansatz eingebunden wurden. Mit Hilfe dieser Transformationen werden SCCharts zuerst in eine textuelle Repräsentation, den SCL Code, transformiert. Auf Basis des grafischen Äquivalents zu SCL, dem SCG, können Analysen durchgeführt werden, mit denen bestimmt werden kann, ob es für ein gegebenes SCL Programm ein gültiges Schedule gibt. Sofern ein solches existiert, kann das SCL Programm mit den gewonnenen Informationen aus den Analysen in einer zweiten Transformation in ein sequentielles SCL Programm ohne Nebenläufigkeiten transformiert werden.

Auf Basis der sequentiellen SCL Programme wurden zwei Transformationen zur Hardwarebeschreibungssprache VHDL präsentiert. In der ersten Transformation wurde der SCL Code direkt, unter der Verwendung des Prozess-Konstruktes, in VHDL übersetzt. Das Prozess-Konstrukt wurde verwendet, da es die Möglichkeit bietet, innerhalb von VHDL sequentiell zu programmieren.

In einer zweiten Transformation wurden die vorhandenen Datenabhängigkeiten innerhalb eines SCL Programms durch die Transformation in eine SSA-Form aufgelöst. Dazu wurde eine Einführung in das SSA gegeben und die Besonderheiten, die bei der Transformation von SCL Modellen bestehen, wurden erläutert. Abschließend wurde eine weitere Transformation vorgestellt, die SSA SCL Code in eine VHDL Datei übersetzt.

Im Folgenden wurden Möglichkeiten zum Testen vorgestellt, die sicherstellen sollen, dass die Transformationen korrekt funktionieren. Zu Beginn wurde ein manueller Test mit Hilfe des Xilinx Tools ISim dargelegt, indem eine ESO Testdatei in eine VHDL-Testbench transformiert wurde, um damit später das transformierte Modell gegen die

Testbench zu validieren. Da diese Art zu testen jedoch sehr aufwändig ist, wurde ein automatischer JUnit Test entwickelt, der in Unterkapitel 5.3 präsentiert wurde. Der Test transformiert selbständig SCCharts in VHDL und dazugehörige ESO Testdateien in eine VHDL-Testbench. Anschließend wird eine Simulation mit den beiden Dateien gestartet und das Ergebnis ausgewertet. Die Ergebnisse aller getesteten Modelle werden zusammengefasst präsentiert.

Anschließend wurden einige Details zur Implementierung gegeben. Unter anderem wurde die Organisation der einzelnen Transformationen in Eclipse Plugins und Details zu ihrer Funktionsweise vorgestellt. Weiterhin wurde eine Einführung zum internen Aufbaus von FGPAs gegeben und eine Implementierung auf realer Hardware präsentiert.

Abschließend wurden eine Evaluation vorgestellt, die die naive und die SSA-basierende Transformation miteinander vergleicht. Es konnte festgestellt werden das die SSA-basierende Transformation tendenziell Schaltkreise mit einer geringeren Chipauslastung und einer geringeren Periodendauer aus SCCharts synthetisieren kann, als die naive Transformation.

Die vorgestellten Lösungen ermöglichen eine Transformation von SCCharts in einen deterministischen Schaltkreis, der das selbe Verhalten wie das SCChart aufweist. Die Evaluierung lässt zusätzlich vermuten, dass die SSA-basierende Transformation Schaltkreise mit einer geringen Chipauslastung und einer kleineren Periodendauer, im Gegensatz zu naiven Transformation, erzeugt. Die in dieser Arbeit gezeigte Lösung hat noch Potential für verschiedene Erweiterungen. Einige mögliche Erweiterungen werden im folgendem Kapitel aufgeführt.

8.2 Weiterführende Arbeiten

Eine allgemein akzeptierte These in der Forschung ist, dass die Forschung niemals endet [Car12]. Während der Entwicklung dieser Arbeit wurden einige Erweiterungsmöglichkeiten festgestellt, die im Folgenden vorgestellt werden.

8.2.1 Datentypen

Bei der vorgestellten Umsetzung wurde in VHDL nur der Datentyp Boolean verwendet. SCCharts hingegen weisen weitere Datentypen wie Integer, Float und String auf. Die Umsetzung für Integer sollte ähnlich wie für den Typ Boolean erfolgen. Es wird in VHDL der Datentyp Integer unterstützt. Die Transformation sollte demnach zu erweitern sein. Zum Testen kann weiterhin das ESO bzw. Core ESO Format genutzt werden. Im ESO Format werden wertbehaftete Signale unterstützt, die im Core ESO Format in zwei Variablen, eine Status- und eine Wert-Variablen, aufgeteilt werden. Die Transformation von wertbehafteten Signalen ist bereits in der ESO zu Core ESO und in der Core ESO zu Testbench Transformation implementiert.

Weitere Formate wie String und Float wären ebenfalls denkbar. Jedoch muss für Strings nach einer entsprechenden Repräsentation in VHDL gesucht werden. Ebenso stellt sich das Problem bei Float ein. Floating Point Operationen werden nicht direkt von VHDL unterstützt, es müssten spezielle Floating Point Recheneinheiten entwickelt werden, um Operationen auf Float Variablen ausführen zu können.

8.2.2 Terminierung

Die Ausführung eines SCCharts terminiert, wenn in allen nebenläufigen Regionen der Kontrollfluss in einem finalen Zustand ist. Es werden keine weiteren Ausgaben mehr berechnet. Die Ausgaben bei einem synthetisierten Schaltkreis ändern sich nicht mehr. Der Schaltkreis selbst jedoch, kann nicht terminieren. Es ist dem Schaltkreis von außen nicht anzusehen, wann das Kontrollflussende erreicht ist. Somit ist nicht klar, ob die derzeitigen Ausgaben zu einer aktuellen Berechnung eines Ticks gehören oder, ob das Kontrollflussende bereits erreicht ist. Es könnte ein weiteres Ausgangssignal term erzeugt werden, welches die Terminierung, bzw. das Erreichen des Kontrollflussendes des Schaltkreises anzeigt. Werte, die die Schaltung produziert, wenn das term-Signal gesetzt ist, könnten somit als ungültig angesehen werden.

Die Implementierung müsste voraussichtlich auf SCL-Ebene stattfinden. Auf dieser Ebene geschieht die Erzeugung des Kontrollflusses, somit kann dort ermittelt werden, wann der gesamte Kontrollfluss terminiert. Beim Erzeugen der sequentiellen SCL-Variante könnte eine weitere Variable gesetzt werden, die anzeigt, dass der Kontrollfluss beendet und das SCL Programm terminiert ist.

Der Wert dieser Variablen kann dazu verwendet werden, den Status des Signals term zu steuern. Es muss darauf geachtet werden, dass das term-Signal erst im darauffolgenden Tick erscheint, denn die aktuellen Ausgaben, die parallel zur Terminierung generiert werden, sind noch gültig. Das term-Signal müsste mit einem Register um einen Takt verzögert werden.

8.2.3 SSA Optimierungen

Die SSA-Transformation transformiert alle Variablen in eine versionierte Instanz, ungeachtet ihrer Nutzung. Guard Variablen werden zum Beispiel nur ein einziges Mal innerhalb eines sequentiellen SCL Programms geschrieben. Dennoch werden für diese Variablen Versionen angelegt. Es könnten weitere Analysen auf dem Code angewendet werden, um unnötige Versionierungen von Variablen zu Gunsten der Lesbarkeit wieder zu entfernen. Eine Konvention, dass alle Variablen, die mit einem g beginnen, guard Variablen sind, und diese dann nicht versioniert werden, erschien zu restriktiv. So müssten weitere Analysen nach der eigentlichen SSA Transformation ausgeführt werden, die unnötige Versionierungen wieder entfernen. Vermutlich würden diese Optimierungen die One-Pass-Transformation komplexer gestalten, hier sollten Analysen die Vor- und Nachteile herausstellen.

8.2.4 Optimierte Transformation

Während der Entwicklung dieser Arbeit wurde weiter am SCL Meta Modell geforscht. So entstand die Idee, die Logik und die Register auf hierarchischer Ebene zu trennen. Die Register sind der Teil der Schaltung, in dem sich der Zustand des SCCharts widerspiegelt. Die Logik beinhaltet den reaktiven Teil einer Schaltung wie UND-, ODER-Gatter und Multiplexer. In Listing 8.1 ist ein ABO SCL Programm aufgelistet, in dem der Zustand des SCCharts und die Logik getrennt ist. Die Trennung wird durch zwei einzelne Module realisiert. Im ersten Modul werden die persistenten Informationen vorgehalten und das *Logik Modul* wird dort aufgerufen. Das reaktive logische Modul ist das ABO_tick_logic Modul in dem Listing 8.1. Diese hierarchische Trennung kommt ebenfalls der Lesbarkeit und Übersichtlichkeit zu Gute. So befinden sich in dem Logik Modul keine zeitlichen Komponenten mehr, zeitliches und reaktives Verhalten können somit voneinander getrennt betrachtet werden.

In der Synthese eines VHDL Schaltkreises könnte die Logik in einer eigenen Entity gekapselt sein. Diese Logik Entity wird als Komponente der VHDL Datei geladen, die alle Register, Sampling und Pre-Register enthält. In Abbildung 8.2 ist der synthetisierte Schaltkreis abgebildet, er beinhaltet die Logik Komponente abo_logic.

Durch ein erweitertes Interface der äußersten Komponente (Abbildung 8.2 abo:1) könnten explizite Werte in die Register geladen werden. Dies ermöglicht, synthetisierte SCCharts in jeden beliebigen Zustand zu versetzen, wenn die Belegung der Register zu dem gewünschten Tick bereits bekannt ist. Diese Idee, kann für die Entwicklung interessant sein: SCCharts könnten während der Entwicklungsphase in jeden beliebigen Zustand versetzt werden, ohne dass alle Ticks bis zum gewünschten Tick ausgeführt werden müssten. Dieser Vorteil könnte sich jedoch angesichts des Umstandes relativieren, dass die Taktfrequenzen von FPGAs bei einigen Megahertz liegen und in jedem Takt ein Tick ausgeführt werden kann. So könnten bei einem Takt von 10MHz, theoretisch 10.000.000 Ticks pro Sekunde ausgeführt werden.

```
1  module ABO_tick
2  input output boolean A;
3  input output boolean B ;
4  input RESET;
5
6  // outputs are always static!
7  output boolean O1 ;
8  output boolean O2 ;
9
10 // Registers
11 static boolean RUNNING, g4, g6, O1, O2;
12 {
13    if RESET then
14    {
15       RUNNING = false;
16       g4 = false;
17       g6 = false;
18       O1 = false;
19       O2 = false;
20    }
21    else
22    {
23 //  RUNNING = RUNNING_pre;
24 //  g4 = g4_pre;
25 //  g6 = g6_pre;
26 //  O1 = O1_pre;
27 //  O2 = O2_pre;
28       //Pseudo Code for Running
29       //ABO_tick_logic module
30       run ABO_tick_logic(A/A, B/B, ...);
31
32 //  RUNNING_pre = RUNNING;
33 //  g4_pre = g4;
34 //  g6_pre = g6;
35 //  O1_pre = O1;
36 //  O2_pre = O2;
37    }
38 }
```

```
55 module ABO_tick_logic
56 input output boolean A, B ;
57 input output boolean O1, O2 ;
58 input output boolean RUNNING ;
59 input output boolean g4, g6 ;
60 boolean g0, g1, e2, e6, g2 ;
61 boolean g3, g5, g7, g8, g9 ;
62 {
63    g0 = ! RUNNING;
64    RUNNING = true;
65    if g0 then
66       O1 = false;
67       O2 = false;
68    end;
69    g5 = g4;
70    g7 = g6;
71    g2 = g0 || g5;
72    g3 = g2 && A;
73    if g3 then
74       B = true;
75       O1 = true;
76    end;
77    g4 = g2 && ! A;
78    g8 = g7;
79    g9 = g8 && B;
80    if g9 then
81       O1 = true;
82    end;
83    g6 = g0 || ( g8 && ! B );
84    e2 = ! ( g4 );
85    e6 = ! ( g6 );
86    g1 = ( g3 || e2 ) &&
87         ( g9 || e6 ) &&
88         ( g3 || g9 );
89    if g1 then
90       O1 = false;
91       O2 = true;
92    end;
93 }
```

Abbildung 8.1. Hierarchischer SCL Code mit zwei Modulen

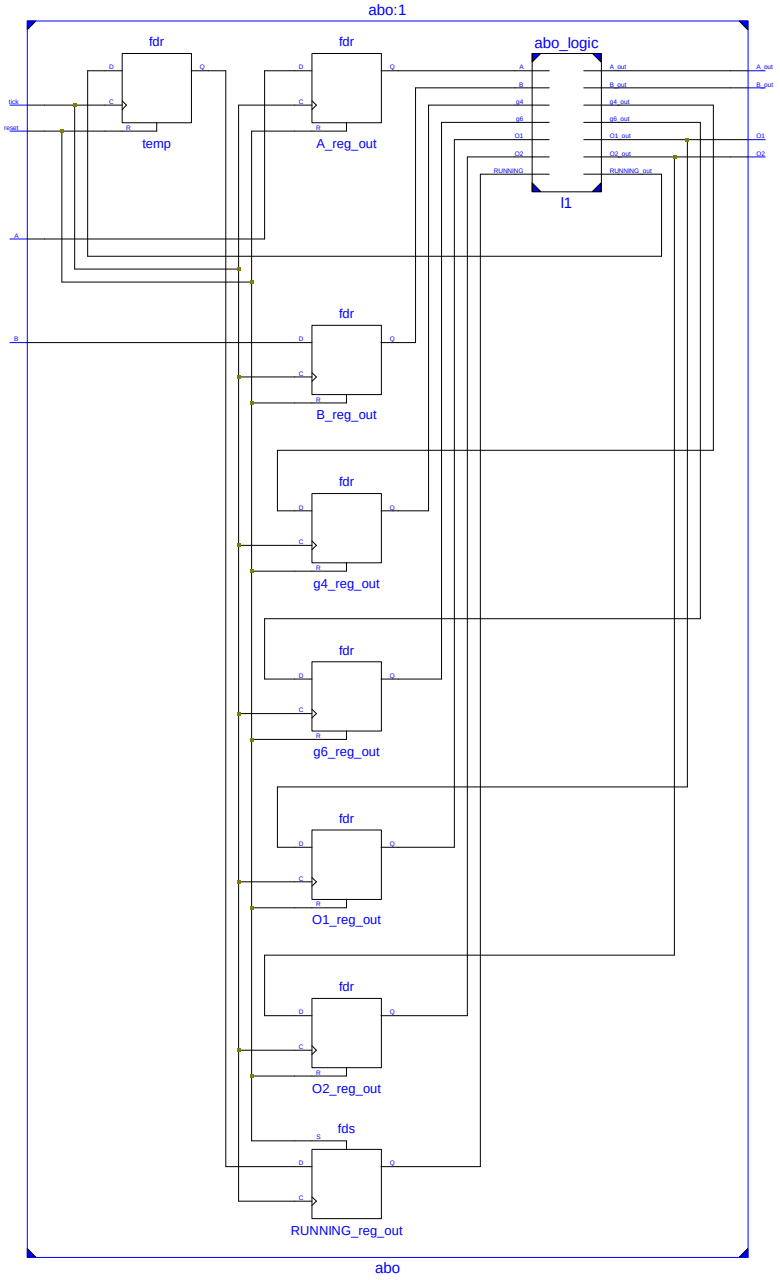


Abbildung 8.2. Hierarchischer Schaltplan von ABO

Danksagungen

Abschließend möchte ich mich noch bei folgenden Personen bedanken, die mir das Schreiben der Masterarbeit ermöglicht und mich während meines Studiums begleitet und unterstützt haben.

Prof. Dr. Reinhard von Hanxleden, Leiter des Lehrstuhls Echtzeit und Eingebettete Systeme, vielen Dank für das Ermöglichen dieser Arbeit und die anregenden Diskussionen und die Betreuung.

Meinen beiden Betreuern Dipl.-Inf. Ass. iur. Insa Marie-Ann Fuhrmann und Dipl.-Inf. Christian Motika, für die umfangreiche Betreuung, den zahlreichen Diskussionen und der hilfreichen Unterstützung.

M. Sc. Ulf Rüegg für die Hilfestellungen in Bezug auf die VHDL Programmierung und die Hardwaresynthese.

B. Sc. Helmut Schimkowski für die gesamte Studienzeit und das Quer-Lesen meiner Arbeit.

Meiner Familie, insbesondere meinen Eltern, Dörte und Hans-Heinrich Johannsen, für die Unterstützung und das Ermöglichen meines Studiums.

Der größte Dank geht an Wiebke Pochert, für die Unterstützung während meines gesamten Studiums, für die Zeit, Energie und Motivation die Sie mir gegeben hat.

Literaturverzeichnis

[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.

[App98] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998. URL: http://doi.acm.org/10.1145/278283.278285, doi:10.1145/278283.278285.

[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.

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

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

[Ber02] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, Version 3.0, December 2002. http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.zip.

[Bha92] J. Bhasker. *A VHDL primer*. BCS Practitioner Series. Prentice Hall, 1992. URL: http://books.google.de/books?id=Wm1TAAAAMAAJ.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, November 1994. URL: http://doi.acm.org/10.1145/197320.197331, doi:10.1145/197320.197331.

[Car12] John Julian Carstens. Node and label placement in a layered layout algorithm. Master's thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2012. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jjc-mt.pdf.

[CN99] J.M.P. Cardoso and H.C. Neto. Macro-based hardware compilation of javatm bytecodes into a dynamic reconfigurable computing system. In *Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on*, pages 2–11, 1999. doi:10.1109/FPGA.1999.803662.

[Cra84] J.D. Crawford. An electronic design interchange format. In *Design Automation, 1984. 21st Conference on*, pages 683–685, 1984. doi:10.1109/DAC.1984.1585881.

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

[DM99] Giovanni De Micheli. Hardware synthesis from c/c++ models. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '99, New York, NY, USA, 1999. ACM. URL: http://doi.acm.org/10.1145/307418.307527, doi:10.1145/307418.307527.

[DMG97] G. De Michell and R.K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997. doi:10.1109/5.558708.

[Edw05] S.A. Edwards. The challenges of hardware synthesis from c-like languages. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 66–67 Vol. 1, 2005. doi:10.1109/DATE.2005.307.

[Est08] Esterel Technologies, Inc. SCADE Suite, last visited 05/2008. http://www.esterel-technologies.com/products/scade-suite/.

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

[GTvH07] Sascha Gädtke, Claus Traulsen, and Reinhard von Hanxleden. HW/SW Co-Design for Esterel Processing. In *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS'07)*, Salzburg, Austria, September 2007.

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

[Har13] Wahbi Haribi. A syncharts editor based on yakindu sct, 2013.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Inc] Xilinx Inc. *VHDL Reference Guide*. http://atlas.physics.arizona.edu/~kjohns/downloads/vhdl/VHDL-xilinx-help.pdf.

[Inc09] Xilinx Inc. *ISim User Guide*, 2009. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/plugin_ism.pdf.

[Inc12a] Xilinx Inc. *Command Line Tools User Guide*, 2012. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/devref.pdf.

[Inc12b] Xilinx Inc. *Synthessis and Simulation Design Guide*, December 2012. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/sim.pdf.

[Inc12c] Xilinx Inc. *VIRTEX-6 FPGA Configurable Logic Block*, 2012. http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.

[Inc12d] Xilinx Inc. *VIRTEX-6 FPGA ML605 EVALUATION KIT*, 2012. http://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf.

[Inc13] Xilinx Inc. *Xilinx — What is a FPGA?*, 2013. http://www.xilinx.com/fpga/index.htm.

[KLM90] David Ku, Stanford University. Computer Systems Laboratory, and Giovanni De Micheli. *HardwareC - a Language for Hardware Design. Version 2.0 -*. Computer Systems Laboratory, Stanford University, 1990.

[KR00] T. Kuhn and W. Rosenstiel. Java based object oriented hardware specification and synthesis. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pages 579–581, 2000. doi:10.1109/ASPDAC.2000.835167.

[LK93] Rachel Y. W. Lau and Hilary J. Kahn. Information modelling of edif. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 278–283, New York, NY, USA, 1993. ACM. URL: http://doi.acm.org/10.1145/157485.164892, doi:10.1145/157485.164892.

[LvH05] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. Technical Report 0509, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, November 2005.

[LvH06] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.

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

[MSvHM13] Christian Motika, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. Sequentially Constructive Charts (SCCharts). Poster presented at 10th Biennial Ptolemy Miniconference (PTCONF'13), Berkeley, CA, USA, November 2013.

[MvHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. Programming deterministice reactive systems with Synchronous Java (invited paper). In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*, IEEE Proceedings, Paderborn, Germany, 17/18 June 2013.

[PBdST05] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. The synchronous hypothesis and synchronous languages. In R. Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005.

[RK13] Realtime and Embedded Systems Group University Kiel. *Kieler Overview Picture*, 2013. http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview.

[RT82] J. Reif and R. Tarjan. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, 1982. URL: http://epubs.siam.org/doi/abs/10.1137/0211007, arXiv:http://epubs.siam.org/doi/pdf/10.1137/0211007, doi:10.1137/0211007.

[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.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, February 2006. doi:10.1109/MC.2006.58.

[SKB02] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. Dynamic power consumption in virtex™-ii fpga family. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, FPGA '02, pages 157–164, New York, NY, USA, 2002. ACM. URL: http://doi.acm.org/10.1145/503048.503072, doi:10.1145/503048.503072.

[SMP88] C.E. Stroud, R.R. Munoz, and D.A. Pierce. Behavioral model synthesis with cones. *Design Test of Computers, IEEE*, 5(3):22–30, 1988. doi:10.1109/54.7960.

[Smy13] Steven Smyth. Code generation for sequential constructiveness, 2013.

[SP98a] 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, 1998. doi:10.1109/FPGA.1998.707944.

[SP98b] D. Soderman and Y. Panchul. Implementing c designs in hardware: a full-featured ansi c to rtl verilog compiler in action. In *Verilog HDL Conference and VHDL International Users Forum, 1998. IVC/VIUF. Proceedings., 1998 International*, pages 22–29, 1998. doi:10.1109/IVC.1998.660676.

[Sta09] Falk Starke. Executing Safe State Machines with the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, January 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fast-dt.pdf.

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

[vH09] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, pages 225–234, Grenoble, France, October 2009. ACM.

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

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

[vHMA+13c] Reinhard von Hanxleden, Michael Mendler, Joaquin 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.