Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Fabian Zahn

Code-Generierung aus SysML-Konnektoren mit gemischten Hardware/Software-Endpunkten

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Fabian Zahn

Code-Generierung aus SysML-Konnektoren mit gemischten Hardware/Software-Endpunkten

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Lehmann
Zweitgutachter: Prof. Dr. Bettina Buth

Abgegeben am 26. August 2011

Fabian Zahn

Thema der Bachelorthesis

Code-Generierung aus SysML-Konnektoren mit gemischten Hardware/Software-Endpunkten

Stichworte

UML, SysML, Codegenerierung, Hardware/Software Co-Design, Kommunikationsschnittstellen, Eingebettete Systeme

Kurzzusammenfassung

In der vorliegenden Arbeit wird untersucht wie in SysML-Modellen beschriebene Hardware/Software-Schnittstellen automatisiert durch Codegeneration synthetisiert werden können. Für diesen Zweck wurde ein Transformationskonzept sowie ein Generatorprototyp entworfen, welcher ein Register File als Hardware-Komponente in VHDL sowie einen Hardware Abstraction Layer für die Software-Komponente erzeugt.

Fabian Zahn

Title of the paper

SysML connector based code generation between mixed hardware/software endpoints

Keywords

UML, SysML, code generation, hardware/software co-design, communication interfaces, embedded systems

Abstract

This paper presents an approach for automated hardware/software interface synthesis based on SysML models. In order to achieve this goal a transformation concept and a prototyp code generator will be shown being capable of generating a register file component in VHDL as well as a hardware abstraction layer for the software module.

”A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools”

– Douglas Adams

Danksagung

An erster Stelle danke ich meiner Familie für ihre Unterstützung während des gesamten Studiums, meiner Freundin besonders für ihre Nachsicht. Nicht zu vergessen all den Kommilitonen, die im Laufe der letzten Jahre zu sehr guten Freunden geworden sind und deren Hilfsbereitschaft und Unterstützung einen nennenswerten Beitrag zum erfolgreichen Abschluss dieses Studiums geleistet haben.

Meinen Betreuern Prof. Dr. Bettina Buth und Prof. Dr. Thomas Lehmann gilt besonderer Dank für ihre Zeit, Geduld und konstruktiven Anregungen.

Ebenso danke ich der Firma Atego für das kostenlose Bereitstellen einer voll funktionalen Lizenz der Modellierungssoftware Artisan Studio.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Durch die steigende Integrationsdichte auf Halbleiterchips ist es möglich immer komplexere eingebettete Systeme auf kleinstem Raum zu entwickeln. Für das Entwerfen von eingebetteten Systemen ergeben sich neue Herausforderungen, um den Ansprüchen gerecht zu werden. Das Design von Schnittstellen zwischen Hardware- und Software-Komponenten ist dabei von großer Bedeutung, da dies ein sehr fehleranfälliger Prozess ist, der nicht nur entsprechend gut entwickelt, sondern auch äußerst detailliert dokumentiert werden muss. Die modellgetriebene Entwicklung dieser Schnittstellen und deren Code-Generierung für Hardware- und Software-Komponenten kann diesen Prozess deutlich vereinfachen und gleichzeitig Fehlerfreiheit garantieren. Ressourcen für die Entwicklung des entsprechenden Systems wären reduzierbar bei gleichbleibender Qualität des Produkts.

1.1 Motivation

Die modellgetriebene Entwicklung hat in der Software-Branche in den letzten Jahren an Bedeutung gewonnen. Einerseits kann man das auf stetig komplexer werdende Softwaresysteme zurückführen, die ein besseres Projektmanagement benötigen. Andererseits leistet die Unified Modeling Language (UML) einen großen Beitrag zur Verbreitung der modellbasierten Ansätze zum Entwurf bzw. zur Entwicklung von Softwareprodukten. Sich daraus ergebende Vorteile sind z.B. die parallele Nutzung menschlicher Ressourcen verschiedener Entwicklungsteams, die sonst sequentiell arbeiten müssten, da Hardware- und Software-Entwicklung meistens voneinander getrennt werden. Außerdem kann durch die modellgetriebene Entwicklung unter Einsatz von Code-Generierung die Zeit eines Produkts von der Spezifikation zur Marktreife verringert werden, da durch Code-Generierung automatisch fehlerfreie Teilprodukte entstehen können. Zudem wird die Übersichtlichkeit des Projekts stark verbessert im Vergleich zum traditionellen dokumentenbasierten Systementwurf. Auch unerfahrene Systemingenieure und Projektmitarbeiter können graphische Modelle wesentlich einfacher verstehen und interpretieren.

1.2 Ziel der Arbeit

Als Produkt dieser Arbeit soll ein Konzept mit einem Code-Generatorprototypen entstehen, welcher den ersten Schritt in die Hardware/Software-Schnittstellen Synthese unter Verwendung von SysML als Modellierungssprache repräsentieren soll. Für diesen Zweck werden Regeln für die Modellierung und die Transformation entwickelt und festgelegt, die die automatisierte Erzeugung von einer Hardware-Abstraktionsschicht und Register File ermöglichen. Sowohl das Konzept als auch der Code-Generator sollen möglichst flexibel ausgelegt werden, damit das System potentiell um neue Plattformen erweitert werden kann. Für den Anwender soll auf die Hardware eine konsistente objektorientierte Sicht entstehen, die den Aufwand zur Ansteuerung der Hardware reduziert.

1.3 Aufbau des Dokuments

In den folgenden Kapiteln wird der Einsatz der modellgetriebenen Entwicklung von Hardware/Software-Schnittstellen untersucht. Für diesen Zweck wird die graphische Systemmodellierungssprache SysML eingesetzt. Deren Mechanismen zur Verbindung von Blöcken unter Verwendung von Ports und Konnektoren werden analysiert und auf ein reales System abgebildet.
Abschließend wird ein Konzept, welches die Transformation vom Spezifikationsmodell zur realen Hardware/Software-Schnittstelle ermöglicht, entworfen und anhand einer Fallstudie verifiziert.

Kapitel 1 - Einleitung
Das erste Kapitel liefert eine kurze Einführung in das Thema, zudem werden die Motivation und das Ziel der Arbeit erläutert.

Kapitel 2 - Grundlagen
In diesem Kapitel werden einige notwendige Kenntnisse referiert, um die Analyse samt ihres Konzepts nachvollziehen zu können. Es werden Grundlagen und Notwendigkeiten von graphischen Modellierungssprachen angeführt und ein kurzer Überblick über den Bereich der Code-Generierung gegeben.

Kapitel 3 - Analyse und Konzept
Der momentane Stand der Technik im Bereich der System- bzw. Schnittstellensynthese wird in diesem Kapitel in Bezug auf Modellierungssprachen und Anwendungsgebiet analysiert. Eingangs wird auf eingebettete Systeme und das Design von Hardware/Software-Schnittstellen eingegangen. Abschließend wird das Konzept zur Modellierung eines durch Code-Generierung synthetisierbaren Systems vorgestellt, wozu die zur Modellierung notwendigen Sprachelemente der SysML angeführt und deren konkrete Bedeutung in der Modellierung des Systems definiert werden.

Kapitel 4 - Realisierung
Anhand einer Feldstudie wird das in Kapitel 3 vorgestellte Konzept implementiert und evaluiert. Des Weiteren werden der Code-Generator und dessen Arbeitsweise detailliert beschrieben. Die vorgestellte Methodik wird unter Verwendung des implementierten Code-Generators verifiziert.

Kapitel 5 - Zusammenfassung und Ausblick
Im letzten Kapitel werden eine Zusammenfassung der Ergebnisse sowie ein Ausblick mit Ideen und Motivationsanregungen für weitere Projekte bzw. die Erweiterung dieser Arbeit gegeben.

1.4 Anmerkung

In dieser Arbeit werden häufig englische Termini für die entsprechenden Fachbegriffe übernommen, da deren deutsche Übersetzungen einerseits nicht im Sprachgebrauch der Domäne akzeptiert werden und andererseits einige technische Begriffe nicht übersetzbar sind ohne ihre Bedeutung zu verlieren.

Außerdem sei an dieser Stelle angemerkt, dass sämtliche Graphiken und Diagramme, die in dieser Arbeit verwendet werden, im Hinblick auf die Weiterverwendung bzw. für Publikationszwecke in englischer Sprache verfasst sind.

Weiterhin sei darauf hingewiesen, dass das in dieser Arbeit verwendete generische Maskulinum weibliche Personen in allen Zusammenhängen ausnahmslos mit einschließt.

2 Grundlagen

Für das Grundverständnis werden im Folgenden graphische Modellierungssprachen, die zur Zeit sehr verbreitet sind, kurz beschrieben und erklärt. Abschließend wird noch eine kurze Einführung in modellbasierte Systementwicklungstechniken und die Code-Generierung gegeben.
Hierzu ist es nützlich sich weiterer Sekundärliteratur zu bedienen, um sich vertiefend in das Thema SysML einzuarbeiten.

2.1 Graphische Modellierungssprachen

Mit graphischen Modellierungssprachen ist es möglich eine abstrakte sowie auch übersichtliche Betrachtung von komplexen Problemen zu entwerfen. Dies gilt unter anderem auch für die Software- und Systementwicklung. Exemplarisch sei ein graphischer Ablaufplan als Flussdiagramm dargestellt, der eine in der Programmiersprache *C* geschriebene Funktion selbst für Menschen ohne Programmierkenntnisse veranschaulichen kann (siehe Abbildung 2.1 und 2.2). Mit der objektorientierten Softwareentwicklung entstand in den 90er Jahren die erste Version der Unified Modeling Language (UML).
Die Sprache setzt sich aus der Zusammenführung verschiedener Notationen und Modellierungssprachen zusammen und wird seitdem stetig weiter entwickelt. Am 19. November 1997 wurde sie der Object Management Group (OMG) übergeben, welche diese als Standard akzeptierte.

2.1.1 Unified Modeling Language: UML™

Die OMG Unified Modeling Language definiert eine allgemein verwendbare Modellierungssprache, die für die Software- und Systementwicklung sowie die Modellierung von Geschäftsprozessen eingesetzt werden kann. Ursprünglich zum Zweck der Spezifikation, Konstruktion und Dokumentation von Softwaresystemen entwickelt, wird sie überwiegend hierfür verwendet.
Die Entwicklung und Standardisierung erfolgen durch die OMG. Durch die Erhebung zum Standard im Jahre 2005 durch die International Organization for Standardization (ISO) ist

```
void count(void){
      int i = 0;
      while(i < 10){
            printf("\ni: %d", i++);
      }
}
```

Abbildung 2.1: C-Code

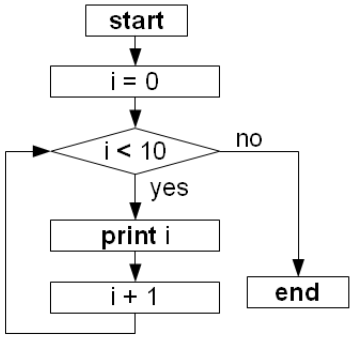

Abbildung 2.2: Flussdiagramm

die UML auch als internationale Norm akzeptiert (UML Version 1.4.2 im ISO/IEC 19501 Standard, siehe [1]).

Die UML definiert Diagramme, Notationen und Semantiken sehr einfach und übersichtlich, womit die verschiedenen Aspekte der entsprechenden Anwendungsgebiete modelliert werden können [6]. Die Sprache ist intuitiv und damit auch für Einsteiger auf dem Gebiet der Modellierung von vielfältigen Systemen geeignet.

Mit Profilen ist es möglich die Sprache sehr flexibel an die verschiedenen Domänen anzupassen. Mit der SysML als Hauptbestandteil dieser Arbeit wird das derzeit prominenteste Profil der UML, das zur Systemmodellierung genutzt wird, verwendet. Die aktuelle Version stellt UML 2.3 (siehe [10]) dar, welche im Mai 2010 von der OMG veröffentlicht wurde.

2.1.2 Systems Modeling Language: SysML™

Die OMG Systems Modeling Language (SysML) ist eine Erweiterung der UML, um die zahlreichen Aspekte der Systemmodellierung, welche von der UML noch nicht abgedeckt wurden, zu integrieren. Zusätzlich wurden die für die Systementwicklung irrelevanten Diagramme und Notationen entfernt [14]. Auch die SysML bietet weiterhin das von der UML bekannte System für Erweiterungen über Profile an, um die Sprache an die spezifischen Anforderungen von Modellierungsdomänen anzupassen. Zudem ist es möglich Modell-Bibliotheken zur Verfügung zu stellen, um bereits entwickelte (Teil-)Systeme für neue Projekte zu übernehmen. In Abbildung 2.3 sind die verschiedenen Diagrammarten der SysML ersichtlich.

Bereits im September 2007 wurde die erste Version der SysML von der OMG veröffentlicht, während die aktuelle Version 1.2 im Juni 2010 erschien. Die Abbildung 2.4 zeigt das Verhältnis zwischen SysML und UML in einem Venn-Diagramm. Die Schnittmenge UML4SysML beschreibt die von der SysML wiederverwendeten Konzepte der UML.

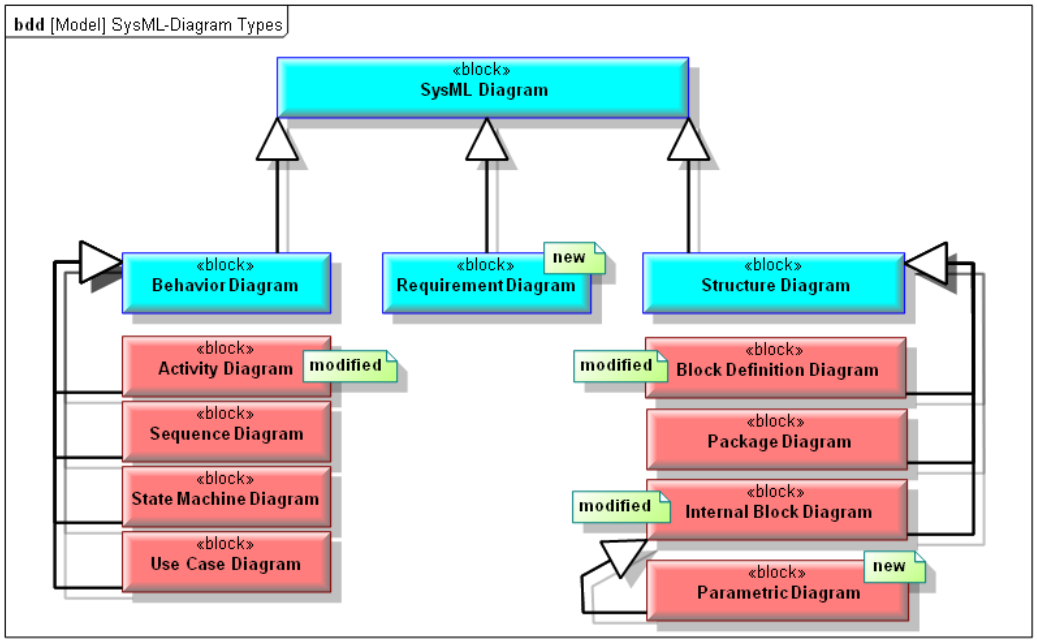

Abbildung 2.3: Die verschiedenen SysML-Diagramme (analog zu [9])

2.1.3 Modeling and Analysis of Real Time and Embedded Systems

Die OMG definiert mit Modeling and Analysis of Real Time and Embedded Systems (MARTE) eine Spracherweiterung für die UML 2, die speziell für die Anforderung der eingebetteten (Echtzeit-)Systeme entworfen wurde. Die erste Version wurde im November 2009 von der OMG veröffentlicht und stellt damit noch eine relativ junge Erweiterung der UML dar. MARTE wurde im Hinblick auf die Modellierung, Analyse, Simulation und Verifikation dieser Systeme entwickelt und bietet die Möglichkeit sowohl Hardware- als auch Software-Aspekte dieser Systeme standardisiert zu modellieren. Momentan ist die Verbreitung von MARTE noch sehr gering und wird zumeist in Forschungsprojekten verwendet.

2.2 Model-Based Systems Engineering

Model-Based Systems Engineering (MBSE) bezeichnet die formalisierte Anwendung der Modellierung, um Systemanforderungen, Design, Analyse, Verifikation und Validierung sowohl zu Beginn der konzeptuellen Entwurfsphase als auch während der Entwicklungsphase und im späteren Lebenszyklus zu unterstützen [3].
Ein Ziel des MBSE-Ansatzes ist es die Kommunikation zwischen den verschiedenen Personen bzw. Teams, welche an der Systemplanung und Entwicklung beteiligt sind, zu verbes-

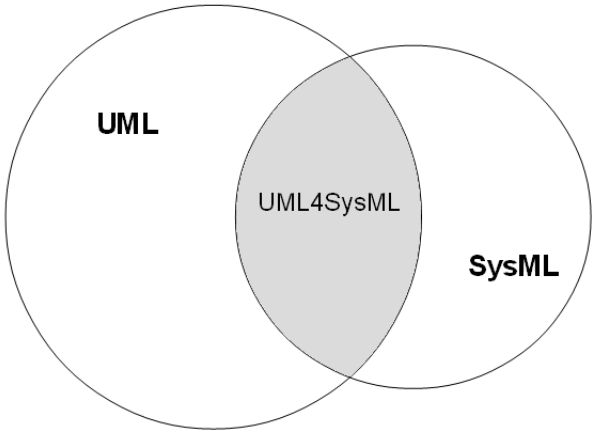

Abbildung 2.4: SysML als Teil und Erweiterung der UML [9]

sern. Außerdem sollen die Produktqualität und die Wiederverwendbarkeit von Systemspezifikationen und Teilsystemen im Vergleich zu traditionellen, dokumentenbasierten Ansätzen verbessert werden. Das Produkt eines durch MBSE entstandenen Systementwurfs soll ein kohärentes Systemmodell liefern, welches durch modellbasierte Methoden und Werkzeuge erweitert bzw. verfeinert wird [2]. Einen guten Einblick in den Systementwurf unter Verwendung des MBSE-Paradigmas bietet die Internetpräsenz der Gesellschaft für Systems Engineering e.V.[1].

2.3 Model-Driven Architecture

Model-Driven Architecture (MDA) ist ein modellgetriebener Ansatz der Systementwicklung basierend auf einer strikten Trennung von Funktionalität und Technik.
Zu diesem Zweck werden die Spezifikationen von Funktionalität und Plattform getrennt (separation of concerns). Es entstehen also immer mindestens zwei Modelle: Ein Plattform-Modell (PM), das die Zielarchitektur definiert sowie ein die Funktionalität beschreibendes, plattformunabhängiges Modell (PIM).
Durch Transformation dieser beiden Modelle kann ein gemeinsames plattformspezifisches Modell (PSM) erstellt werden. Mit diesem Modell wird eine Code-Generierung oder Synthese des Modells ermöglicht.
Anders als einige andere Ansätze zum modellbasierten Entwurf von Software erfordert ein MDA-Ansatz keine hundertprozentige Automatisierung des Systems, lediglich die sinnvoll erzeugbaren Anteile des Systems sollen durch Code-Generierung entstehen.

[1] http://mbse.gfse.de/ (Stand: August 2011)

2.4 Code-Generierung

Als Code-Generierung bezeichnet man die automatische Erzeugung von Quelltexten in einer bestimmten Programmier- bzw. Beschreibungssprache durch einen Code-Generator. Compiler sind die meist genutzten Code-Generatoren, da diese aus Hochsprachen-Code nativen Maschinen- oder Bytecode erzeugen. Weitere Beispiele für Code-Generatoren sind unter anderem auch Decompiler, Assembler oder Programme zur Erstellung von Webseiten. Unter Verwendung eines Code-Generators ist es möglich fehlerfreie Quelltexte aus Modellen, abstrakten Beschreibungen oder formalen Sprachen zu erstellen.

Die Code-Generierung bildet den zentralen Aspekt für die Entwicklungsansätze: Model-Driven Architecture und Computer-Aided Software Engineering (CASE). Diese nutzen die Code-Generierung, um formale Spezifikationen von Modellen in Quellcode zu transformieren. In der Literatur sind viele verschiedene Ansätze zur Code-Generierung zu finden. Eine sehr gute Übersicht über die gebräuchlichen Verfahren kann man in [13] einsehen. Die Abbildung 2.5 zeigt die Code-Generierungsphasen für eine beispielhafte Top-Down Code-Generierung aus einem graphischen Modell.

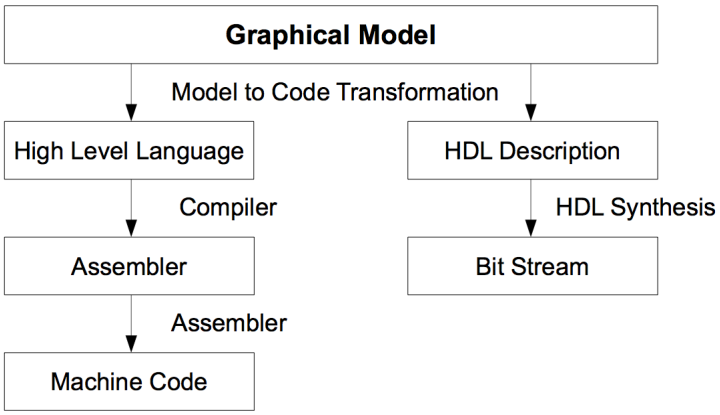

Abbildung 2.5: Top-Down Code-Generierung aus einem graphischen Modell

Die in dieser Arbeit verwendete Code-Generierung setzt sich mit der sogenannten Register File Synthese sowie mit der Generierung einer Hardware Abstraktionsschicht auseinander, die im Folgenden diskutiert wird.

2.4.1 Register File Synthese

Die Register File Synthese (RFS) beschreibt den Prozess, der für die Generierung des Register Files zuständig ist. Als Register File bezeichnet man hier die im Block organisierten

Register, die zum Speichern von Informationen zwischen Mikroprozessor und Hardware-Peripherie genutzt werden.
Das Register File wird in einer Hardware Description Language (HDL) beschrieben und kann unter Verwendung von sogenannten High-Level-Synthesewerkzeugen in konkrete Hardware synthetisiert werden, um z.B. auf FPGA-Implementierungen Verwendung zu finden. Die RFS erzeugt sowohl die einzelnen Register als auch die für die Zugriffe auf die einzelnen Register benötigten Schreib- und Lesedecoder.

2.4.2 Hardware Abstraction Layer

Eine zusätzliche Abstraktionsschicht, die die Sicht auf die Hardware als Register abstrahiert, bezeichnet man als Hardware Abstraction Layer (HAL). Durch einfache und schnelle Methoden (Funktions- bzw. Makroaufrufe) ist es mit dem HAL möglich, komplizierte Bitmanipulationsoperationen auf dem Register File fehlerfrei auszuführen. Dadurch entstehen Vorteile für die Entwicklung, da ein deutlich lesbarer Quellcode entsteht, wenn die Bitmanipulationen verdeckt unter einem Funktions- bzw. Makroaufruf vorgenommen werden. Zusätzlich sind die HAL-Funktionen bzw. Makros in jedem Fall fehlerfrei, wenn eine fehlerfreie Implementierung des Code-Generators gegeben ist.
Somit können die Fehleranalysen eingegrenzt und deren Zeitaufwand minimiert werden.

3 Analyse und Konzept

Beginnend mit der Analyse des aktuellen Stands der Technik in Bezug auf die Synthese von Interfaces und Systemen mit modellbasierten Ansätzen wird im folgenden Kapitel untersucht, wie sich die Hardware/Software-Kommunikation zusammensetzt und was benötigt wird, um eine automatisierte Code-Generierung aus formalen Spezifikationen zu ermöglichen.
Abschließend folgt eine Auswertung in Konzeptform, die sich aus der Analyse ableitet und die signifikantesten Resultate zusammenfasst.

3.1 Analyse

3.1.1 Stand der Technik

In den letzten 10 Jahren sind in Bezug auf die Synthese von Hardware/Software Interfaces unterschiedliche Forschungen durchgeführt worden. Nachfolgend werden drei Projekte angeführt, die sich sowohl mit dem Thema der automatisierten Schnittstellensynthese als auch mit dem computergestützten Entwurf dieser Schnittstellen beschäftigen.

Interface Synthesis (Hardware/Hardware)

Im Rahmen der Dissertation mit dem Titel ”Modeling and Automated Synthesis of Reconfigurable Interfaces” [5] von Stefan Ihmor wurde an der Universtität Paderborn erforscht, wie inkompatible Kommunikationselemente von Hardware-Schnittstellen unter Verwendung eines Adaptermoduls durch automatisierte Synthese miteinander verbunden werden können.
Diese Adaptermodule werden als Interface-Blöcke (IFB) bezeichnet und werden durch einen automatisierten Prozess als synthetisierbarer VHDL-Code erzeugt.

Die Kommunikation kann mit UML State Machines beschrieben werden, wofür das State Machine Diagram mit einer eigenen Semantik und Syntax versehen wird. Die Interface-Blöcke sind zur Laufzeit rekonfigurierbar und können so mit wechselnden Tasks oder Medien verbunden werden.
In der Dissertation wird gezeigt, dass es möglich ist (inkompatible) Hardware-Schnittstellen

durch die Verwendung von Adaptermodulen über modellbasierte Entwurfstechniken zu erzeugen.

ComiX/TempliX (Hardware/Software)

Zur Beschreibung der registerbasierten Kommunikation in eingebetteten Systemen wurde im Jahr 2001 an der Carl von Ossietzky Universität Oldenburg eine XML-basierte Interface-Beschreibungssprache mit dem Namen ComiX entwickelt. Durch die detaillierte Spezifikation der Details, die für eine Registerkommunikation benötigt wird, ist es damit möglich einen Treiber für die Hardware sowie Hardware-Bausteine für die Ein- und Ausgabe zu erzeugen. Der Generator *TempliX* verarbeitet dafür die ComiX XML-Beschreibung des Interfaces sowie eine weitere XML-Beschreibung, welche das Template definiert. Damit ist es möglich mit nur einem Generator unterschiedliche Generate zu erzeugen (siehe Abb. 3.1).
Nachteilig zu bewerten ist jedoch das Erlernen der für die Templates benötigten XML-Beschreibungssprache. Die Lesbarkeit des Templates bzw. dessen Beschreibung wird dadurch stark eingeschränkt und lässt eine Fehlerüberprüfung nur schwer zu.

Mit den drei existierenden Template-Sets ist es möglich Hardware-Treiber in der Sprache Ada95 zu generieren, das Register File in VHDL zu beschreiben sowie eine Dokumentation der Kommunikation in Latex zu erzeugen. Die Sprache Ada95 wurde deshalb als Zielsprache gewählt, weil die Repräsentation der einzelnen Objekte im Speicher exakt beschrieben werden kann. Der Compiler ist danach dafür zuständig die korrekte Reihenfolge der Bytes im Speicher (Endianess) und die Selektion von N-Bits für einen Datentyp durchzuführen.
In vielen anderen Sprachen (wie z.B. C) ist dies nicht möglich und muss bei der Code-Generierung bedacht werden. Für den Entwurf der Kommunikation als ComiX-Interface-Spezifikation wurde im Rahmen der Dissertation von Frank Oppenheimer[11] ein Software-Werkzeug mit dem Namen Deshico entworfen (siehe Abb. 3.2) . Dieses sollte mit graphischen Elementen den textuellen Aufbau des Register File Mappings in der ComiX-Spezifikation visuell ermöglichen und als Folge dieser Anschaulichkeit das Finden und Entfernen von Fehlern in der Entwurfsspezifikation vereinfachen.

Folglich wurde durch dieses Projekt ein Ansatz zum Entwurf von Hardware/Software-Schnittstellen vorgestellt, indem eine auf der XML basierte Beschreibungssprache eingesetzt wurde, um die detaillierte Registerkommunikation zu beschreiben.

MoPCoM (Hardware/Software)

Mit MoPCoM wurde von französischen Forschern im September 2006 ein Projekt ins Leben gerufen, um eine Co-Design-Methodik für die Modellierung eingebetteter Systeme

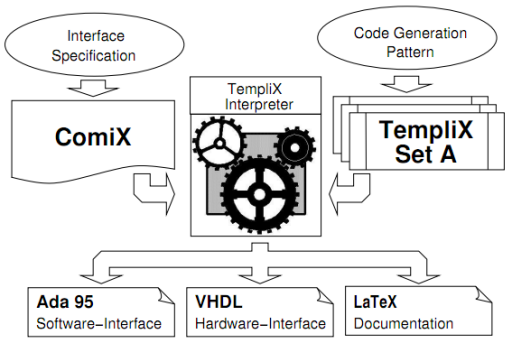

Abbildung 3.1: Comix Flow [11]

zu entwerfen. Sie benutzt MARTE Modelle, um System-on-Chip (SoC) bzw. System-on-Programmable-Chip (SoPC) Systeme zu entwerfen. Für diesen Zweck werden drei aufeinander aufbauende Modellierungsebenen definiert, die das Modell jeweils verfeinern, um letztendlich eine Code-Generierung aus dem finalen Modell zu ermöglichen.
MoPCoM nutzt dafür die aus der MDA und MDE bekannten Ansätze, um die Systemsynthese und Schnittstellensynthese als Produkt der Methodik zu erhalten. Um dies zu erreichen, werden verschiedene Stereotypen eingeführt, damit definierte Plattformen und Interfaces für die Modellierung von Systemen zur Verfügung stehen. Eine sehr gute Übersicht dieses Modellierungsansatzes findet sich in Abbildung 3.3. Für die Generierung von Quellcode aus MoPCoM-Modellen kann man deshalb sehr gut Verfahren einsetzen, die auf vordefinierten Bibliotheken basieren.
Weitere Ziele der MoPCoM-Methodik sind es, eine verbesserte Dokumentation des Projekts zu schaffen und die Analyse des Systems bereits im Modell durchführen zu können.
Mit MoPCoM wird dem Anwender ein sehr umfangreicher Ansatz zum Entwurf und zur Verifikation von Hardware/Software-Systemen unter Verwendung von Paradigmen des modellbasierten Entwurfs von Software bzw. Systemen zur Verfügung gestellt.

OSSS/fossy

Oldenburg System Synthesis Subset (OSSS) stellt eine Erweiterung zu den SystemC Standard-Bibliotheken dar und definiert gleichzeitig eine Methodik zum Entwurf von eingebetteten Systemen auf einer höheren Beschreibungsebene (siehe Abb. 3.4). Das Forschungsprojekt wurde vor 10 Jahren am OFFIS Institute for Information Technology an der Carl von Ossietzky Universität Oldenburg von Wolfgang Nebel und Frank Oppenheimer ins Leben gerufen.

Die OSSS-Methodik setzt beim Entwurf von eingebetteten Systemen auf einen Top-Down-Ansatz, d.h. zuerst wird ein sogenanntes ”Golden Model”, welches vollständig in *C++* be-

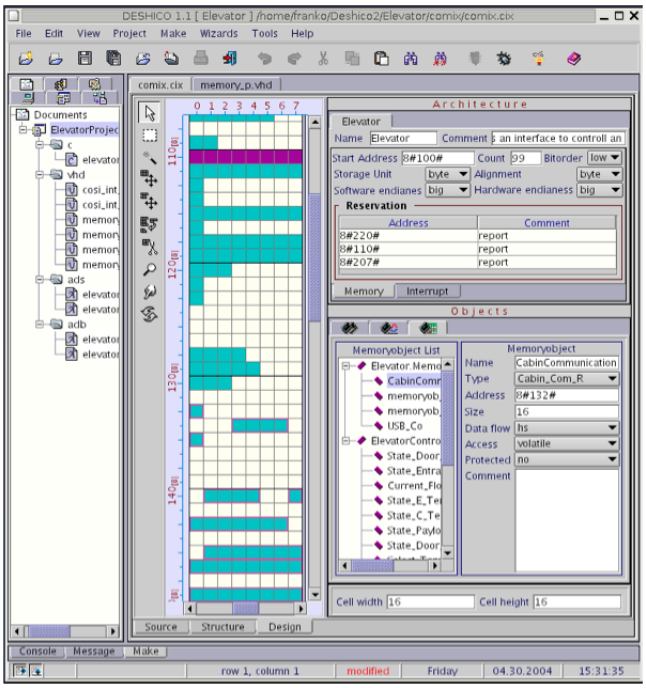

Abbildung 3.2: Deshico Werkzeug [11]

schrieben wird, entworfen. Anschließend wird dieses Eingabemodell verfeinert, indem die verschiedenen strukturellen Elemente identifiziert, klassifiziert und voneinander getrennt werden. Diese strukturellen Elemente können nun in SystemC-Modulen und sogenannten OSSS Software Tasks beschrieben werden. Die Kommunikation und Synchronisation zwischen den verschiedenen Elementen werden über OSSS Shared Objects realisiert. Diese sind spezielle SystemC-Klassen, die über definierte Methodeninterfaces verfügen.
Durch die Verwendung dieser Kommunikationsobjekte zur Schnittstellenbeschreibung ist es möglich einen konsistenten Zugriff von einer beliebigen Anzahl von nebenläufigen Prozessen zu modellieren [12].
Mit fossy wurde im Rahmen dieses Forschungsprojektes ein Werkzeug entworfen, dass die Transformation von der SystemC-Beschreibung aus dem OSSS-Ansatz in synthetisierbaren VHDL Code erlaubt. Außerdem soll durch fossy ein nahtloser Prozess bereitgestellt werden eingebettete Hardware/Software-Systeme unter Verwendung der OSSS-Bibliotheken zu entwerfen und zu synthetisieren (siehe Abb. 3.5).

Dieses Projekt zeigt, dass es möglich ist, vollständige eingebettete Systeme automatisch aus Modellbeschreibungen zu erzeugen, und diese anschließend mit eigens dafür entworfenen sowie proprietären Software-Werkzeugen auf einer Zielplattform zu realisieren.

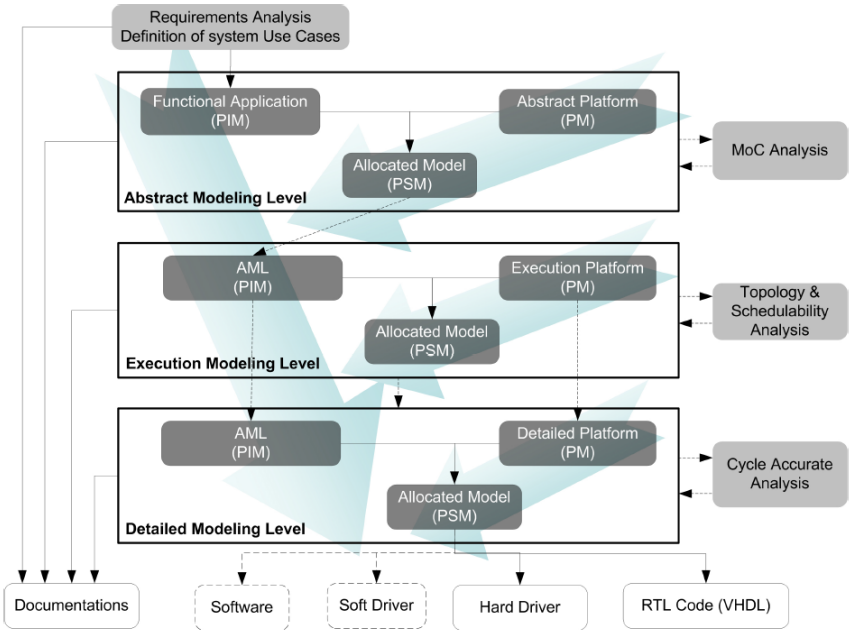

Abbildung 3.3: Der MoPCoM-Modellierungsansatz [7]

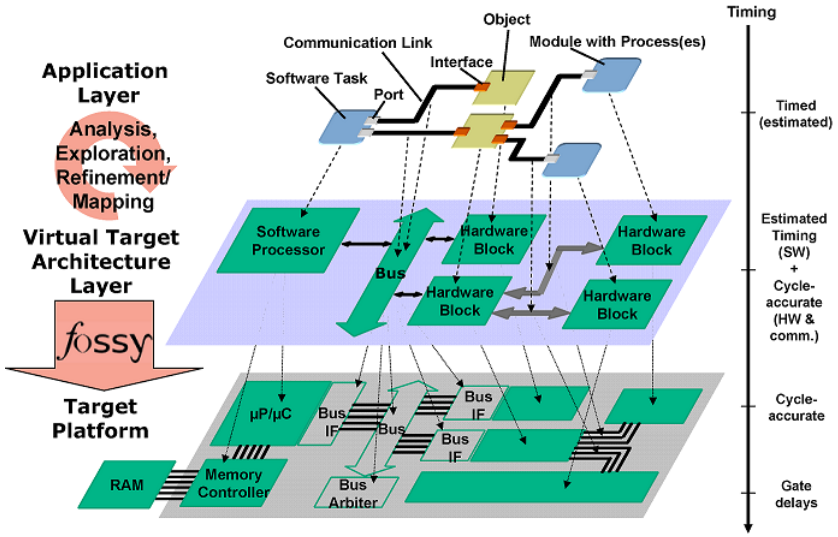

Abbildung 3.4: OSSS-Methodik [12]

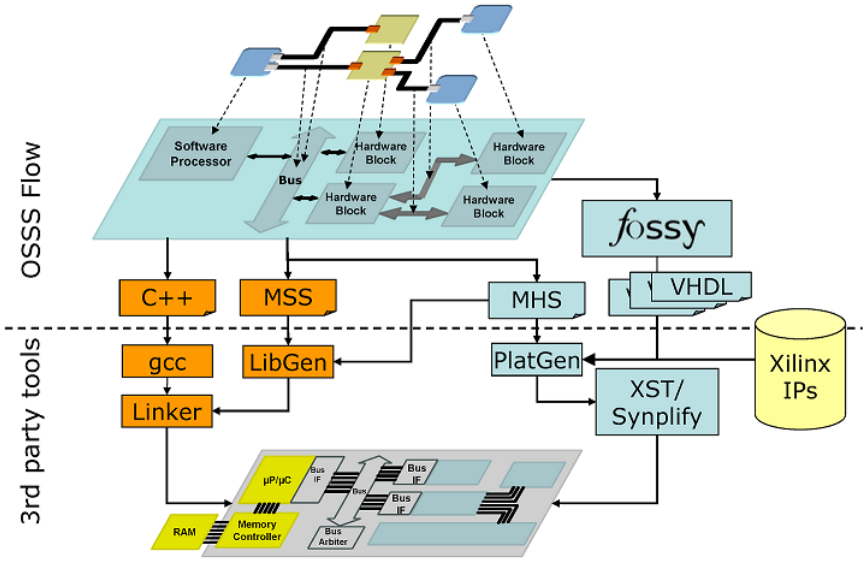

Abbildung 3.5: Der fossy-Syntheseablauf [12]

3.1.2 Eingebettete Systeme

Eingebettete Systeme bestehen meist aus Hardware- und Software-Komponenten, die in ein System integriert und somit weitestgehend unsichtbar für den Benutzer sind. Sie werden verwendet, um das einbettende System zu steuern, zu regeln oder zu überwachen und können so vielfältig eingesetzt werden.
In nahezu jedem modernen Haushaltsgerät oder Multimedia-Equipment existiert mindestens eines dieser Systeme, häufig sogar mehrere, die zusammen ein verteiltes komplexes Gesamtsystem bilden.
Aufgrund des Moorschen Gesetzes und des stetig größer werdenden Design-Productivity-Gaps in der Entwicklung von eingebetteten Systemen werden bessere Methoden benötigt, um eingebettete Systeme zu entwerfen.
Aus diesem Grund ist es ein Ziel dieser Arbeit ein System zu entwickeln, das aus formalen Modellen eine automatisierte Schnittstellensynthese durchführt. Angemerkt sei dabei, dass die in dieser Arbeit untersuchten eingebetteten Systeme in jedem Fall als Komposition aus Hardware- und Software-Komponenten aufzufassen sind, da die Entwicklung von Software/Software- und Hardware/Hardware-Schnittstellen bereits in anderen Arbeiten untersucht wurde.

3.1.3 Hardware/Software Interfaces

Der Entwurf von Hardware/Software-Schnittstellen ist für die Entwicklung von eingebetteten Systemen von großer Bedeutung, da die Realisierung einer Schnittstellenspezifikation ein fehleranfälliger und zeitaufwändiger Prozess ist.
Zum Austausch von Daten zwischen den Software- und Hardware-Komponenten werden Register in der Hardware-Peripherie eingesetzt, die meist über Memory Mapped I/O angesprochen werden können. Zudem existiert in wenigen Fällen noch der Zugriff über Port Mapped I/O, der jedoch nicht sehr geläufig ist.
Die Zusammenfassung dieser Register werden im Allgemeinen als Register File bezeichnet. Demzufolge wird das Register File der Hardware-Peripherie, unter Benutzung von Memory Mapped I/O, direkt in den Adressraum des Mikroprozessors gemappt und kann so mit einfachen Zeigeroperationen adressiert und manipuliert werden.
Auf diese Weise wird das zu Grunde liegende Bussystem abstrahiert und für den Programmierer unsichtbar.
Darüber hinaus existieren auch Systeme, die dedizierte Bustreiber benutzen, um Informationen mit der Hardware-Peripherie auszutauschen. Der Bustreiber stellt Funktionen oder Makros zur Verfügung, die den Zugriff auf die Register Files der verschiedenen Hardware-Endpunkte ermöglichen. Generell lässt sich die Register File basierte Kommunikation zwischen Hardware- und Software-Komponenten in eingebetteten Systemen über das in Ab-

bildung 3.6 dargestellte Blockschaltbild veranschaulichen. Die sogenannten Interrupts, die asynchrone Hardware-Ereignisse signalisieren, sind dabei kein Bestandteil dieser Kommunikation und müssen extern beschrieben werden.

Die für die allgemeine Kommunikation zu übertragenden Informationen zwischen Hardware- und Software-Endpunkten lassen sich in vier Kategorien aufteilen. Die Kategorie einer Nachricht kann durch deren Flussrichtung und Typ bestimmt werden (siehe Abb. 3.7).

- **Steuerungsinformationen**, die von der Software ausgehend genutzt werden, um den Kernel der Hardware zu steuern.
- **Statusinformationen** sind Nachrichten, die von der Hardware aus gesendet werden, um über den internen Status der Hardware Aufschluss zu geben.
- **Daten** werden verwendet als Oberbegriff für die konkreten Daten, welche zwischen Hardware und Software (z.B. analog/digital gewandelte Eingangsdaten einer Messgröße) versendet werden.
- **Interrupts** sind zuständig für das (asynchrone) Mitteilen eines Ereignisses des Hardware Kernels bzw. für das Anfordern von Rechenzeit zur Verarbeitung der von der Hardware bereitgestellten Daten.

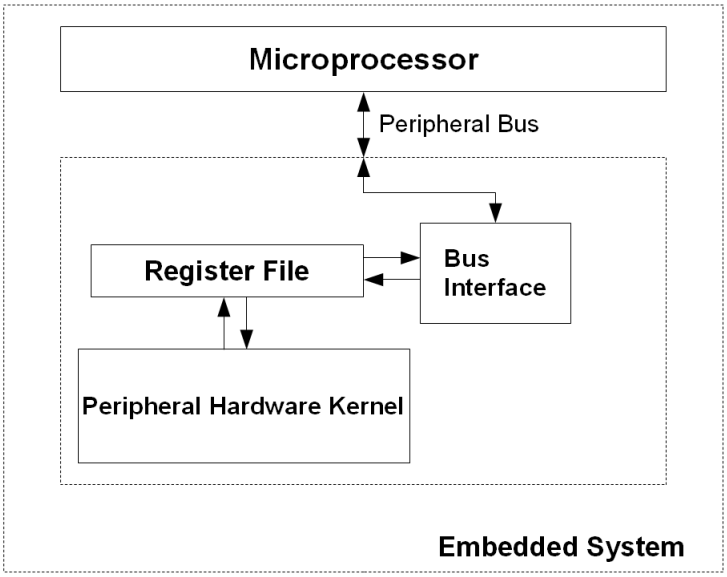

Abbildung 3.6: Register File basierte Hardware/Software-Kommunikation

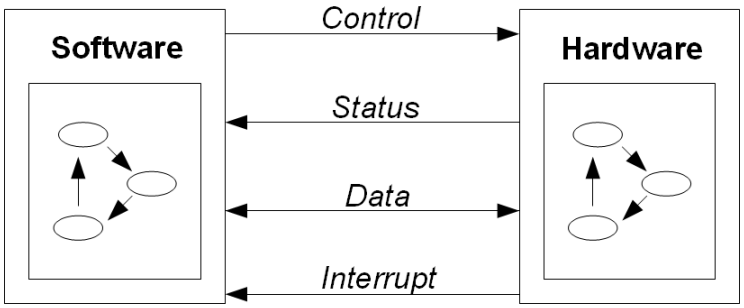

Abbildung 3.7: Die allgemeine Hardware/Software-Kommunikation

3.1.4 Kommunikationsmodell

Das zu Grunde liegende Kommunikationsmodell von Hardware/Software-Schnittstellen lässt sich sehr gut in drei Schichten zerlegen (siehe Abb. 3.8). Dabei lassen sich Parallelen zum ISO/OSI-Schichtenmodell ziehen, bei dem die einzelnen Schichten als jeweilige Erweiterung der vorherigen Schicht aufeinander aufbauen.
In diesem Fall spezialisieren die einzelnen Schichten die vorhergehende und erhöhen damit deren Abstraktionslevel.

Auf der untersten Ebene (der **Bitmanipulationsschicht**) werden Bitmanipulationen vorgenommen, d.h. einzelne Bits werden sowohl von der Hardware- als auch von der Software-Komponente des Systems manipuliert, um die verschiedenen Steuerungs-, Status- und Dateninformationen auszutauschen. Der Low-Level-Treiber für die Hardware-Peripherie stellt eine Software-Realisierung dieser Ebene dar.

Die darauf folgende Schicht ist die **Typisierungsschicht**, die die einzelnen Bits zu konkreten Nachrichten zusammenfasst. Man beschreibt auf welche Datentypen die einzelnen Nachrichten abgebildet werden sollen und definiert Mengen für einzelne Objekte aus deren Wertebereich gewählt werden darf, um Informationen zu senden oder zu empfangen. Die Realisierung dieser Schicht in Software stellt die nächsthöhere Abstraktion des Low-Level-Treibers dar, den Hardware Abstraction Layer.

Die höchste Abstraktion der Hardware-Peripherie bildet die **Applikationsschicht**. Datenobjekte werden dahingehend konkretisiert, dass eine explizite Interpretation der Daten entsteht. Beispielsweise wird ein 8 Bit breiter Datentyp als 8.2 Festpunkt Zahl interpretiert, also eine Dezimalzahl in Binärcodierung mit 6 Bits für die Stellen vor dem Komma und 2 Bits für die Nachkommastellen. Die Ausgangsdaten eines Temperatursensors können somit direkt in einem definierten Format ausgelesen werden.

Als Produkt der automatisierten Code-Generierung werden in dieser Arbeit die beiden untersten Schichten in Form eines Hardware Abstraction Layers generiert werden. Dem

SysML-Modell müssen zweckentsprechend Informationen annotiert werden, um die zur Code-Generierung benötigten Details dieser Kommunikation zu modellieren.

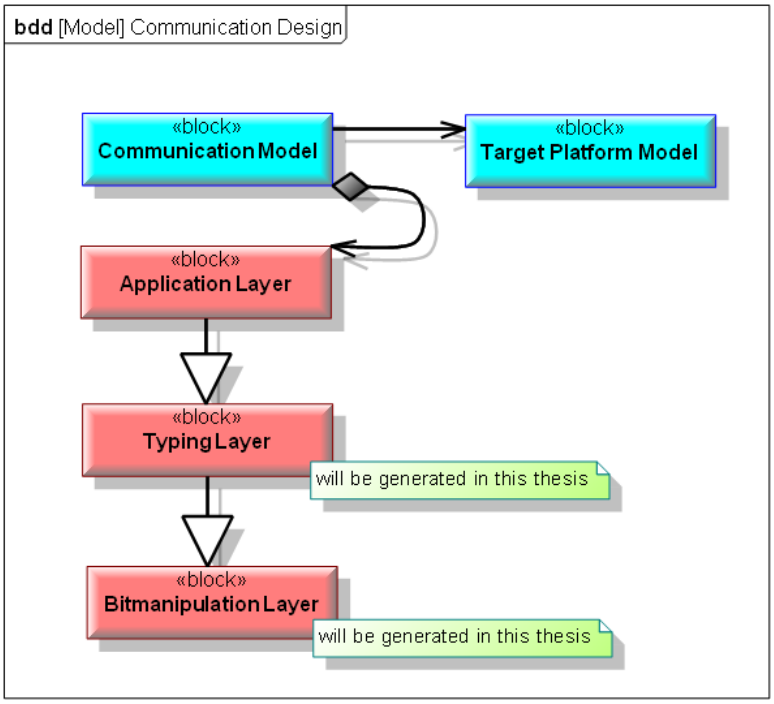

Abbildung 3.8: Veranschaulichung des Kommunikationsmodells

3.1.5 Hardware/Software Co-Design

Wie der Name bereits vermuten lässt, ist Hardware/Software Co-Design ein interdisziplinärer Ansatz zum Entwurf von Systemen, die sowohl Hardware- als auch Software-Komponenten umfassen.

In der traditionellen Entwicklung von Systemen entstand der Entwurf dieser zwei Domänen sequentiell. Die Hardware wurde spezifiziert und entworfen, um anschließend eine Software-Schicht auf das System aufzusetzen. Zur Vermeidung von Problemen beim Entwurf dieser immer komplexer werdenden Systemen, wurde in den letzten Jahren verstärkt an Co-Design-Ansätzen geforscht, um zusammen, koordiniert und nebenläufig an komplexen Systemen zu arbeiten. Die Zeit für die Entwicklung eines Systems lässt sich durch den Einsatz von Hardware/Software Co-Design verkürzen, da theoretisch die Ingenieursressourcen der Fachgebiete Hardware- und Softwaredesign parallel genutzt werden könnten, wie in [11] deutlich hervorgehoben wird. Ferner wird gesichert, dass der Informationsaustausch von nicht trivialen Details zwischen den Domänen bereits beim Entwurf des Systems gegeben ist. Verstän-

digungsproblematiken und eventuelle Missverständnisse können dadurch direkt beim Design sowohl erkannt als auch beseitigt werden.

Bei sequentieller Arbeit ist dies nicht für jeden Fall garantiert, da bereits in Produktion gegebene Hardware naturgemäß nicht immer vollständig anpassbar ist.
Ein modellbasierter Ansatz zur Spezifikation bzw. Entwicklung von Hardware/Software-Schnittstellen kann demnach, durch eine gemeinsame Entwicklungsplattform, dazu beitragen diese Defizite zu verringern. Durch die gemeinsame Entwicklung an definierten Spezifikationen kann der Entwicklungsprozess optimiert werden.
Die Popularität und Akzeptanz der UML ist in der Softwareentwicklung bereits gegeben und bildet deshalb eine gute Grundlage für die Adaption der SysML als Entwurfssprache. Ebenso sind auch in der Hardwareentwicklung modellbasierte Entwurfsverfahren bereits etablierte Standards. Folglich ist die Integration von SysML zur Spezifikation und Entwicklung von Systemen in beiden Domänen ohne große Schwierigkeiten möglich.

3.1.6 Field Programmable Gate Array

Als Field Programmable Gate Array (FPGA) bezeichnet man spezielle, integrierte Schaltkreise (IC), in die logische Schaltungen programmiert werden können. Ein FPGA wird mit einem Programm beschrieben, welches jedoch eher einer Strukturbeschreibung von digitaler Hardware entspricht als einem Programm im konventionellen Sinn, so dass es einen anwendungsspezifischen Zweck erfüllt. Diese Programme können sehr einfache logische Grundschaltungen, aber auch komplexe eingebettete Systeme mit Mikroprozessoren umfassen.

Aus diesem Grund sind FPGAs momentan sehr beliebt, um Entwicklungskonzepte zu verifizieren oder konkrete Hardware-Prototypen zu testen, bevor diese z.B. als anwendungsspezifische integrierte Schaltungen (engl. Application Specific Integrated Circuit: ASIC) in die Massenfertigung gehen.
Ebenso gewinnen FPGAs immer größeren Einfluss auf dem Markt, da neben Plattformen für Rapid Prototyping auch stetig mehr Endkundengeräte FPGAs enthalten (u. a. USB Oszilloskope oder AV Receiver). FPGAs sind sehr flexibel einsetzbar und können durch Updates sogar Fehler in der Hardware nachträglich beheben.
Für militärische Anwendungen werden FPGAs gerne genutzt, weil diese ihre Programmierung ohne Versorgungsspannung verlieren und deshalb für den Feind keinen weiteren Aufschluss über die interne Technologie geben können.
Aufgrund dieser Flexibilität und Konfigurierbarkeit bietet sich ein FPGA für die Entwicklung eines Prototypsystems für diese Arbeit idealerweise an.

3.1.7 Soft-Prozessoren

Ein Soft-Prozessor (oder auch Softcore-Mikroprozessor) ist ein in einer Hardwarebeschreibungssprache entwickelter Mikroprozessor, der flexibel durch Logiksynthese in programmierbaren Logikbausteinen eingesetzt werden kann. Viele Hersteller von programmierbarer Logik bieten bereits fertige Soft-Prozessoren an, die speziell an die konkreten Chips des Herstellers angepasst wurden.

Daneben existiert inzwischen jedoch eine große Community im Internet, die Open Source Soft-Prozessoren und andere Open-Source IP Cores zur Verfügung stellt: OpenCores[1].

Die bekanntesten Soft-Prozessoren sind zurzeit der Xilinx Microblaze und der NIOS II von Altera. Mittlerweile existieren auch ausgereifte binärkompatible Open Source Klone dieser Prozessoren, die ohne Lizenzgebühren verwendet werden können.
Die beiden Prozessoren verfügen über Bussysteme, die zu einer Erweiterung flexibel genutzt werden können, indem zusätzliche Hardware-Peripherie oder Co-Prozessoren in das System integriert werden. Für diesen Zweck existiert bereits eine große Auswahl an fertigen IP Cores, die von den Herstellern teils gratis, teils gegen Lizenzgebühren zur Verfügung gestellt werden.
Die Hersteller von FPGAs bieten in der Low-Cost-Sparte selten eine auf dem FPGA integrierte Hardcore-Prozessorlösung an, da diese Einsteiger-FPGA-Chips jedoch mit vielen Ressourcen für programmierbare Logik ausgestattet sind, bietet es sich an, Soft-Prozessoren auf diesen Chips einzusetzen.
Diese Einsteigerprodukte der FPGA-Sparte lassen häufig die Integration von mindestens einem, mitunter auch zwei Soft-Prozessoren zu und sind dadurch fähig leistungsstarke (Multi-) Prozessorsysteme zu stellen.

3.1.8 Periphere Bussysteme

Prozessoren bieten meistens mehrere Bussysteme an, um das System zu erweitern. Neben Adress- und Datenbus existiert im Allgemeinen mindestens ein lokaler Bus, an den die interne Hardware-Peripherie angebunden wird. Beispielhaft ist dafür der PCI-Express-Bus moderner Computer über den der PC, durch das Hinzufügen von Steckkarten, erweitert werden kann.
Für den Entwurf eingebetteter Systeme auf SoPC-Basis bietet sich ein lokaler Bus an, um das eingebettete System mit zusätzlicher, aus Hardware bestehender Peripherie zu erweitern. Obwohl weiterhin das Paradigma gilt, dass alles, was in Software gelöst werden kann, auch mit Software gelöst werden sollte, gibt es einige Ausnahmen.

[1] http://www.opencores.org (Stand: August 2011)

Das Encoding bzw. Decoding von Videos z.B. ist in Hardware wesentlich performanter möglich als es mit Software der Fall wäre. Sobald funktionelle oder Performanzeinbußen, die durch eine Software-Realisierung entstehen würden, existieren und diese Einbußen für das Projekt nicht mehr vertretbar sind, lohnt es sich dedizierte Hardware einzusetzen.

Der Zugriff auf die, über den lokalen Bus angeschlossene Peripherie, kann auf mehrere Arten erfolgen. In den meisten Fällen wird jedoch Shared Memory verwendet, das über Memory Mapped I/O angesprochen werden kann. Das Register File der Hardware-Peripherie stellt dabei das Shared Memory dar und wird direkt in den Adressraum des Mikroprozessors eingebunden (gemappt). Das Register File ist damit über einfache Zeigeroperationen oder Strukturen, die auf diesen Speicherbereich gelegt werden, zugänglich.

Zurzeit werden die folgenden Standard-Bussysteme eingesetzt, um Soft- oder Hardcore-Prozessoren um zusätzliche Peripherie zu erweitern.

- **On-Chip Peripheral Bus (OPB)** – Ein von der Firma Xilinx verwendetes Bussystem, welches auf IBM CoreConnect basiert. Dieser Bus wird nur noch in älteren Systemen verwendet und gegenwärtig durch den PLB verdrängt.
- **Peripheral Local Bus (PLB)** – Das aktuelle Bussystem der Firma Xilinx dient dazu, den Microblaze Soft-Prozessor und den PowerPC Hardcore-Prozessor mit Hardware-Peripherie zu verbinden.
- **Advanced eXtensible Interface (AXI)** – Das zukünftige Standard-Bussystem der Firma Xilinx, welches bereits in neuen Designs verwendet wird. Die AXI-Spezifikation ist Teil der Advanced Microcontroller Bus Architecture (AMBA), eine Entwicklung der britischen ARM Limited. Auf Xilinx-Geräten kann diese Architektur erst seit der 6. FPGA-Generation (Spartan-6, Virtex-6) eingesetzt werden.
- **Avalon** – Das Bussystem des Herstellers Altera für die NIOS II Soft-Prozessoren.
- **Wishbone** – Ein unter einer Open Source Lizenz stehender Bus, der in vielen OpenCores-Projekten Verwendung findet.

3.1.9 Register File Design

Das Register File bildet die Schnittstelle für den Kommunikationskanal und damit ist es naheliegend zu untersuchen, welche Implementierungsarten existieren und wie diese zu bewerten sind.
Der Zugriff auf das Register File von der Software-Seite erfolgt über ein Bussystem. Aus diesem Grund bestimmt die Breite (in Bit) des Busses unmittelbar die Breite eines Registers ($B_{\text{Bus}} = B_{\text{Register}}$).
Grundsätzlich existieren für den Entwurf von Register Files verschiedene Ansätze, die sich in zwei Gruppen aufteilen: zum einen die strukturellen Unterschiede und zum anderen die funktionalen Unterschiede des Designs.

Der Struktur nach kann man das Register File wiederum auf zwei verschiedene Arten entwerfen, nämlich durch die Benutzung von sogenannten Dual-Ported RAM-Blöcken oder durch das Zusammensetzen des Register Files aus diskreten Flip-Flops.
Das Dual-Ported RAM hat den Vorteil, dass zwei Lese- oder Schreiboperationen nahezu zeitgleich ausgeführt werden können, jedoch eine zusätzliche Arbitrationslogik nötig ist, um Kollisionen oder Zugriffskonflikte aufzulösen. Nachteilig ist hierbei eine ungenaue Vorhersagbarkeit darüber, was tatsächlich bei gleichzeitigem Schreibzugriff von Hardware und Software im Register File vor sich geht. Einen weiteren Nachteil des Dual-Ported RAM ist die daraus entstehende Plattformabhängigkeit, da je nach FPGA-Hersteller über verschiedene Makros für den Dual-Ported RAM, die dedizierten Block-RAM-Ressourcen instantiiert werden.

Die Zusammensetzung des Register Files aus Flip-Flops bietet sich aus diesen Gründen für eine plattformunabhängige Lösung an, da die Beschreibung von Flip-Flops in einer HDL-Beschreibung herstellerübergreifend übereinstimmt.

Die funktionellen Unterschiede beziehen sich auf die Operationen, die auf das Register File ausgeführt werden können. Zur Gewährleistung von atomaren Zugriffen auf die einzelnen Bits des Registers kann dieses beispielsweise über zwei Adressen angesprochen werden, um ein Clear bzw. Set auf ein einzelnes Bit durchzuführen.
Nachteilig bei dieser Lösung ist, dass im Adressraum des Mikroprozessors die doppelte Adressanzahl für den Zugriff auf das Register File benötigt wird.
Ein weiterer Ansatz zur Realisierung von atomaren Zugriffen auf das Register File ist über das strikte Trennen der verschiedenen Objekte auf dem Kommunikationskanal möglich. Diese werden für diesen Zweck, wie bereits beschrieben, in Kontroll-, Status- und Dateninformationen partitioniert. Aus dieser Trennung sind unidirektional gerichtete Beziehungen zwischen den Kommunikationsendpunkten des Register Files ableitbar (siehe Abb. 3.9). Sie ermöglichen es, dass jeweils nur ein Endpunkt in die entsprechende Partition des Register Files schreiben kann. Der Lesezugriff ist weiterhin auf alle Register von beiden Seiten möglich.

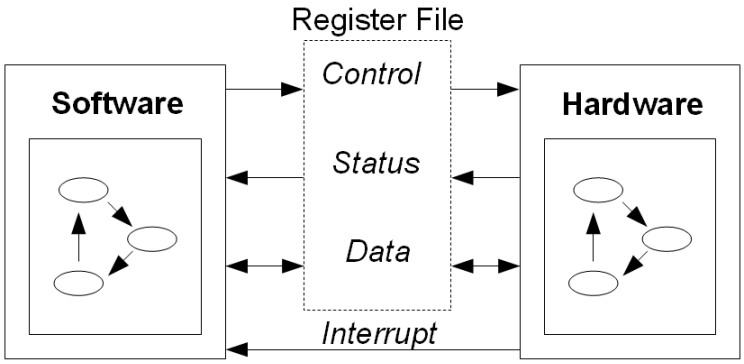

Abbildung 3.9: Register File basierte Hardware/Software-Kommunikation

3.1.10 Verwendung der SysML zur Schnittstellenmodellierung

Zur Übersicht der in dieser Arbeit verwendeten Sprachelemente der SysML erfolgt eine Analyse der Einsetzbarkeit zur Modellierung der Hardware/Software-Schnittstellen. Zusätzlich werden einige elementare Sprachelemente erklärt, um das Konzept durchdringen zu können.

Stereotypen

Die SysML erlaubt es Erweiterungen an der Sprache vorzunehmen. Dies geschieht unter Verwendung von Stereotypen, mit denen neue Definitionen und Erweiterungen an vorhandenen Sprachelementen vorgenommen werden können. Deren Kennzeichnung erfolgt durch Einfassen des Stereotypennamens in Guillemets.
Durch diesen Mechanismus ist es beispielsweise möglich, dass sich ein Hersteller von FPGAs ein vollständiges SysML-Profil für seine IP Cores erstellt. Diese so zur Verfügung gestellten IP Cores können hierdurch für die Modellierung eines digitalen Systems mit Standardkomponenten genutzt werden. Für Designer und Entwickler ließe sich dadurch eine sehr gute Beschreibung des Systems schaffen, die bereits während des Entwurfs einen Aufschluss über die Zieltechnologie gibt.
Ein anderer möglicher Ansatz wäre es zu versuchen nur die vorhandenen Sprachelemente der SysML für die Systemmodellierung zu benutzen. Hierbei besteht weiterhin eine allgemeine und abstrakte Sicht auf das System und eine deutlich höhere Portabilität des Modells in Bezug auf andere Plattformen bzw. Implementierungen.
Generell sollte man neue Stereotypen nur einführen, wenn dies unvermeidbar ist. Die bereits vorhandenen Sprachelemente sind äußerst vielfältig einsetzbar, so dass meistens kein Bedarf besteht neue Stereotypen zu definieren.

Pakete

Zur Organisation von Modellen werden Paketdiagramme genutzt, darüber hinaus können mit diesen Diagrammen weiterreichende Spracherweiterungen (Profile) der SysML definiert werden. Hierzu werden Stereotypen, Typdefinitionen oder Einheitendefinitionen in einem Profilpaket organisiert. Ein Paket für Typdefinitionen könnte z.B. aus den Integer-Datentypen des ANSI-C99 Standards zusammengesetzt werden, um diese in Modellen bereitzustellen. Umfangreiche Modelle lassen sich ohne Pakete nicht übersichtlich darstellen, da im Allgemeinen zu viele verschiedene Blöcke in einem Modell existieren.

Blöcke

Ein System wird über seine Systembausteine definiert, die wiederum in der SysML als Block bezeichnet werden. Blöcke sind universell einsetzbar und können z.B. zur Beschreibung von Hardware, Software und Anlagen dienen.
Die Kennzeichnung für einen Block erfolgt über das Stereotyp «block». Für alle Strukturdiagramme der SysML bildet der Typ Block den zentralen, beliebig erweiterbaren Datentyp.

Enumeration

Auch die SysML bietet mit Enumerationen einen Aufzählungstyp an. Unter Verwendung dieses Typs kann ein mit endlichem Wertebereich definierter Datentyp geschaffen werden. Der Wertebereich ist unmittelbar über die einzelnen Bezeichnungen der Werte gegeben.
Im Gegensatz zu den geläufigen Programmiersprachen besitzen diese Bezeichnungen der Werte keine vordefinierten diskreten Zahlenwerte. Vielmehr werden in dieser Arbeit die Zahlenwerte unter Verwendung einer Kodierungs-Annotation innerhalb eines Kommentars implizit angegeben, so dass jedem Namen genau ein Wert implizit zugewiesen wird.

Blockdefinitionsdiagramm: bdd

Aus der UML bekannte Klassendiagramme werden durch die SysML zur Definition der Systemhierarchie zu Blockdefinitionsdiagrammen ausgebaut. Sie gehören zur Gruppe der Strukturdiagramme. Diese Black-Box-Sicht wird z.B. zur Darstellung von Beziehungen und Abhängigkeiten zwischen Blöcken verwendet, ohne deren interne Funktionsweise detailliert zu beschreiben.
Abbildung 3.10 zeigt ein exemplarisches eingebettetes System, das aus drei Komponenten besteht. Die Verbindung von Mikroprozessor und Hardware-Peripherie ist über ein Bussystem gegeben.

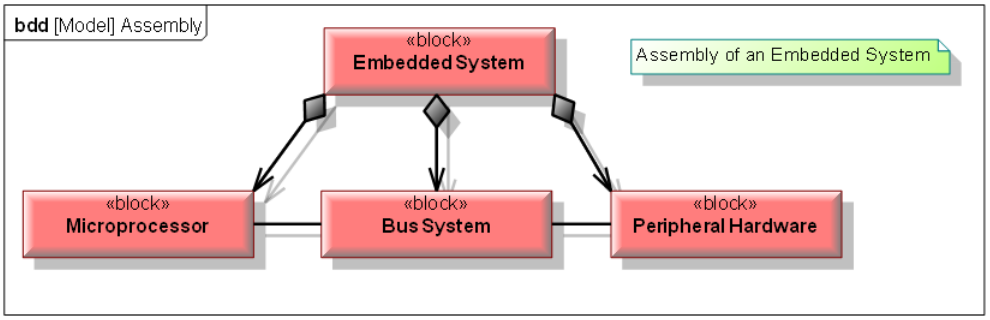

Abbildung 3.10: Ein Realitätsausschnitt eines aus Komponenten zusammengesetzten eingebetteten Systems

Internes Blockdiagramm: ibd

Interne Blockdiagramme (ibd) bilden eine Erweiterung zu den aus der UML bekannten Kompositionsstrukturdiagrammen. Mit ihnen wird die innere Struktur von Blöcken modelliert. In diesen Diagrammen werden Teilsysteme sowie deren Schnittstellen und Verbindungen modelliert, als deren Ergebnis man die sogenannte White-Box-Sicht des Blocks erhält.
Zur besseren Veranschaulichung dieses Konzepts wurde ein fiktiver Signalgenerator in Abbildung 3.11 modelliert.

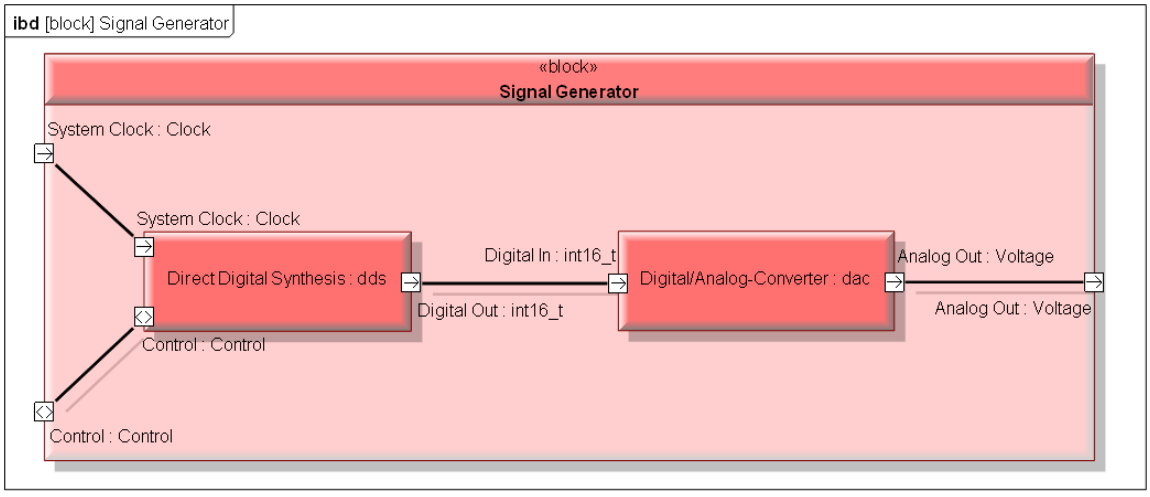

Abbildung 3.11: Das interne Blockdiagramm eines fiktiven Signalgenerators

Ports

Mit Ports werden in der SysML die Interaktionspunkte zwischen Blöcken spezifiziert, wovon in der SysML zwei Arten existieren. Neben den bereits aus der UML bekannten Standard Ports sind noch die sogenannten Flow Ports als neues Sprachelement der SysML hinzugefügt worden.
Mit Ports können im Allgemeinen Schnittstellen für Blöcke zur Verfügung gestellt werden, um Kommunikations-, Objekt- oder Materialflüsse zu modellieren. Eine Verbindung von kompatiblen Ports wird durch die Verwendung von Konnektoren hergestellt.

Standard Ports

Verschiedene Services können genau wie in der UML auch für die SysML unter Verwendung von Standard Ports bereit gestellt werden. In dieser Arbeit werden die Standard Ports als Mittel zur Modellierung des Interrupt-Signals genutzt, da diese Signale eher einem Service als einem Informationsfluss entsprechen. Das Interrupt-Signal trägt für Prozessoren, die nur eine Interrupt-Leitung besitzen, keinerlei Information. In der Interrupt Service Routine (ISR) muss erst über das Abfragen von Flags herausgefunden werden, welche Quelle den Interrupt ausgelöst hat.

Flow Ports

Flow Ports gehören zu der Klasse der Objektflussports mittels derer Daten-, Materialien- oder Energieflüsse modelliert werden. Man unterscheidet grundsätzlich zwischen zwei verschiedenen Arten von Flow Ports, atomaren und nicht atomaren Ports.
Die atomaren Flow Ports bieten die Möglichkeit genau einen Objekttyp über den Port fließen zu lassen. Hingegen kann für nicht atomare Flow Ports mit sogenannten Flow Specifications («FlowSpecification») eine Menge von Objekttypen, die über diesen Port fließen, inklusive deren Flussrichtung spezifiziert werden.
Ein nicht atomarer Flow Port, dessen Flow Specification nur einen Objekttyp beinhaltet, ist jedoch zu einem atomaren Flow Port äquivalent. Die Kompatibilität von Flow Ports ist durch die korrekte Verwendung von Art, Typ und Richtung gegeben. Ein nicht atomarer Flow Port wird mit einem dazu konjugierten Flow Port verbunden, um eine konsistente Verbindung zwischen diesen Ports, die dieselbe Flow Specification benutzen, herzustellen. Die Konjugation bezieht sich dabei auf die implizite Inversion der Flussrichtung, welche in der dazugehörigen Flow Specification angegeben ist.
Eine Übersicht der verschiedenen Port-Typen ist aus der Abbildung 3.12 ersichtlich.

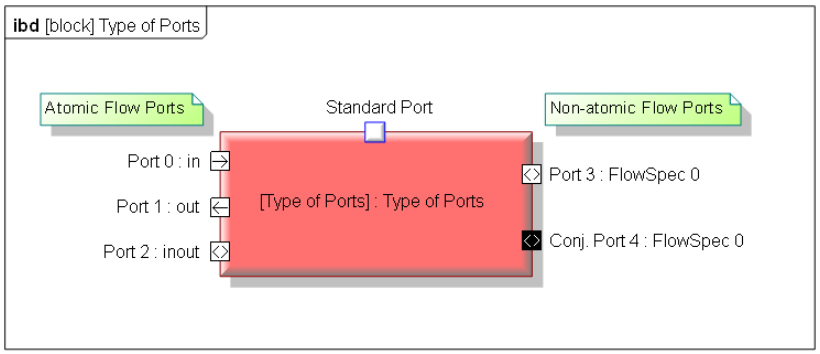

Abbildung 3.12: Die verschiedenen Port Typen der SysML

Flow Specifications

Nicht atomare Flow Ports werden, wie bereits oben erwähnt, über Flow Specifications detaillierter beschrieben. Für diesen Zweck wird einem Flow Port diese Flow Specification zugewiesen und die für die Kommunikation benötigten Objekte, die über diesen Port fließen sollen, werden als Flow Properties («flowProperties») der Flow Specification hinzugefügt. Die Flow Property legt den Namen, den Typ und die Richtung eines über den Flow Port zu übertragenden Objekts fest. Der Typ hierfür kann extern modelliert werden und beispielsweise eine Enumeration oder einen Block annehmen.

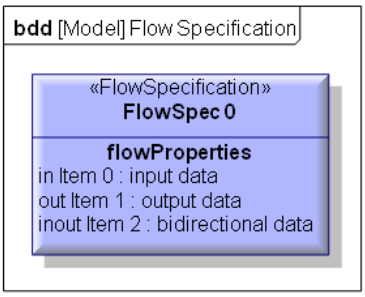

Abbildung 3.13: Exemplarische Flow Specification

3.2 Gesamtkonzept

Bedingt durch die Analyse ergibt sich ein Produkt, dass das Gesamtkonzept für diese Arbeit bildet. Die im vorigen Abschnitt beschriebenen Teile, die zur Code-Generierung von Hardware/Software-Schnittstellen nötig sind, lassen sich nun zusammenfassen, um daraus ein Konzept zu erstellen, dass diese Transformation vom formalen Modell zur realen Schnittstelle ermöglicht.

Es werden zwei **Modelle** zur Spezifikation der vollständigen Kommunikation zwischen Hardware/Software-Endpunkten verwendet. Ein Plattform-Modell, zur Spezifikation der Zielarchitektur, sowie ein Kommunikationsmodell zur eigentlichen (abstrakten) Definition der Schnittstelle. Der Entwurf dieser beiden Modelle ist obligatorisch und muss bestimmten Transformationsregeln folgen.

Zur **Modellierung** der Schnittstelle werden Flow Ports für die Kontroll-, Status- und Datenobjekte verwendet. Interrupt-Signale werden mit den Standard Ports der SysML beschrieben. Für den Entwurf von Schnittstellen sind nur die komplexen Kommunikationen, die Flow Specifications verwenden, von Bedeutung. Atomare Flow Ports können jederzeit auf diesen abgebildet werden und bedürfen aus diesem Grund keines eigenen Konzepts.
Die Repräsentation von bidirektionalen Flow Properties in einer Flow Specification ist zerlegbar in jeweils eine *in* und eine *out* Property und kann in der Modell-zu-Modell-Transformation vorgenommen werden.

Im Rahmen der **Code-Generierung** wird in dieser Arbeit die Typisierungsschicht des hier verwendeten Kommunikationsmodells generiert. Hierfür wird ein Hardware Abstraction Layer in der Sprache *C* sowie ein Register File als VHDL-Beschreibung generiert.
Die Kodierungen für Kontroll- und Statustypen werden dafür automatisiert erzeugt.
Zur Gewährleistung eines nahezu atomaren Zugriffs auf das Register File und um Kollisionen zu vermeiden, wird eine lokale Trennung der Daten im Register File vorgenommen.
Als Produkt des Vorganges der Code-Generierung entsteht eine objektorientierte Sicht auf das konkrete Register File und damit die Hardware-Peripherie.

Das **Register File** stellt die konkrete Realisierung der Hardware/Software-Schnittstelle dar und wird aus diskreten Flip-Flops anstelle von dedizierten Block-RAM-Komponenten aufgebaut, um eine maximale Plattformunabhängigkeit zu erreichen.
Die einzelnen Register des Register Files werden unidirektional (für Schreiboperationen) verwendet und benötigen deshalb keine doppelte Adressierung für Set-/Clear-Operationen.

Als **Informationen, die für die Code-Generierung benötigt werden**, bezeichnet man hier die zur Code-Generierung erforderlichen minimalen Anreicherungen, die dem Modell hinzugefügt werden müssen.

Die Systemanforderungen (Requirements) des zu entwerfenden Systems müssen bestimmt werden, um eine Kommunikationsschnittstelle zu definieren. Die Kodierung für die einzelnen Kommunikationsobjekte muss von dem Entwickler explizit angegeben werden, zum Erreichen einer konsistenten Implementierung sowohl auf der Software- als auch auf der Hardware-Seite des Systems.
Ein sehr wichtiger Faktor ist die Endianess (die Reihenfolge der Bytes im Speicher), die sowohl auf der Hardware- als auch auf der Software-Seite angegeben werden muss, da sonst große Probleme für Datentypen, die größer als ein Byte sind, entstehen können.
Die letzte Information, die dem Modell hinzugefügt werden muss, ist die Busbreite in Bit. Die Angabe dieser Breite wird zur Bestimmung der Registerbreite sowie der maximalen Größe von einzelnen Steuerungs- und Statusinformationen genutzt.

4 Realisierung

Zur Verifikation des vorgestellten Konzepts wurde exemplarisch ein Code-Generator entworfen, der aus einer realen Beschreibung einen synthetisierbaren bzw. kompilierbaren fehlerfreien Quellcode erzeugt.
Als Hardware-Endpunkt wurde ein Inkrementalgeber gewählt, welcher durch einen Microblaze Soft-Prozessor der Firma Xilinx gesteuert werden soll.

Obwohl die Hardware/Software-Schnittstelle dieses Prototyps nur von begrenzter Komplexität ist, wird durch dieses relativ einfache Beispiel deutlich, dass das Konzept nutzbar ist. Die Verwendung einer wesentlich komplexeren Schnittstelle als Fallstudie würde die Überschaubarkeit des Projektes reduzieren und zudem nicht dazu beitragen das Konzept noch deutlicher darzustellen.

Resultierend daraus wird die Realisierung des Konzepts anhand der Fallstudie eines Inkrementalgebers weiter beschrieben.

4.1 Inkrementalgeber/Rotary Encoder

Als Inkrementalgeber werden Sensoren zur Erfassung von linearen Lageänderungen oder rotierenden Winkeländerungen bezeichnet. Verschiedene Verfahren ermöglichen die Erfassung von Wegstrecken, Wegrichtungen und Winkelveränderungen.
Inkrementalgeber sind universell einsetzbar und werden z.B. in ABS-Sensoren oder zur Überwachung von drehenden Maschinen verwendet, um Aufschluss über die internen Maschinenzustände zu geben.
Als Quadraturencoder werden Inkrementalgeber häufig bezeichnet, da diese an ihren zwei Ausgangsanschlüssen (A, B) ein sogenanntes Quadratursignal erzeugen. Dabei handelt es sich um zwei periodische Rechtecksignale, die zueinander um 90° verschoben sind (siehe Abb. 4.1). Unter Auswertung der Signale zueinander kann über das vorauseilende Signal auf die Drehrichtung der Achse des Inkrementalgebers geschlossen werden [15].

Zur Evaluation wurde ein Inkrementalgeber der Firma Noble (RE0124) eingesetzt, der manuell, wie ein Potentiometer, verstellbar ist und z.B. für die Einstellung der Lautstärke in Verstärkern verwendet wird.

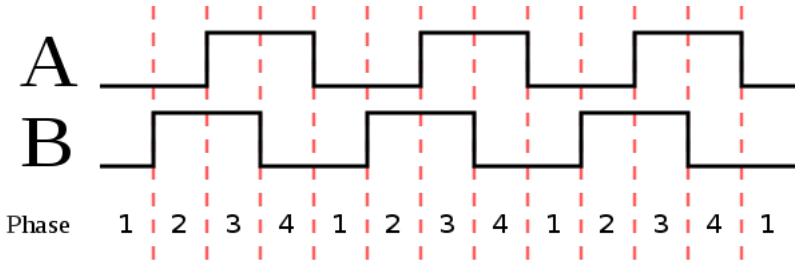

Abbildung 4.1: Idealisiertes Impulsdiagramm Quadratursignal [15]

Aufgrund der Tatsache, dass die Ausgänge des verwendeteten Inkrementalgebers nicht prellfrei sind, werden diese im Kernel des IP Cores entprellt, um die auftretenden Fehler im Eingangssignal zu kompensieren [8]. Das Ausgangssignal des Inkrementalgebers wird im IP Core für diesen Zweck mit 50 MHz abgetastet und anschließend gefiltert.

4.2 Anforderungen

Für den Entwurf einer Hardware/Software-Schnittstelle müssen im ersten Schritt die Anforderungen an die Hardware und die Software festgelegt werden. Diese System Requirements können anschließend von dem Designer der Schnittstelle in konkrete Kommunikationsobjekte übersetzt werden. Nach der Festlegung der Voraussetzungen können diese dann im zweiten Schritt im Hardware Kernel und im Software Back-End implementiert werden. Folglich bildet das Requirement Engineering damit die erste Entwicklungsstufe des Projekts.

Weiterführend sind die speziellen Anforderungen getrennt nach ihren Einsatzbereichen näher beschrieben und ergeben damit das Lastenheft dieser Realisierung.

Allgemeines

- Erfassung der Drehrichtung
- Erfassung der Drehschritte (Ticks)
- Zwei voneinander unabhängige 16-Bit Zähler (für die Drehrichtungen) zur Einstellung eines zu erreichenden Wertes

Steuerung

- Software-seitiges Starten und Stoppen des IP Cores
- Freischalten bzw. Sperren der verschiedenen Interruptquellen des IP Cores

Status

- Abfragemöglichkeit des IP Core Status (laufend oder angehalten)
- Abfragemöglichkeit des Interrupt Status (auslösende Quelle)

Interrupts

- Auslösen eines Ereignisses für Zählerüberläufe
- Auslösen eines Ereignisses zum Erreichen eines definierten Zählerwertes für jede Drehrichtung

4.3 Zielarchitektur

Zur Evaluierung und Verifikation des vorgestellten Konzepts bzw. des implementierten Code-Generators wurde ein FPGA-basiertes Testsystem verwendet, das sich, wie aus der Analyse hervorgeht, ideal als Rapid Prototyping System anbietet.

Evaluation Board

Für die Implementierung der Fallstudie in einem realen System, wurde das FPGA Evaluation Board Nexys2 der Firma Digilent eingesetzt, welches mit einem Spartan 3E (Spartan 3E-1200 FG320) Chip ausgestattet ist (siehe Abb. 4.2).
Für das vorliegende System ist dieser, kapazitiv gesehen, relativ kleine Chip ausreichend. Das verwendete Board bietet viele Möglichkeiten für Erweiterungen an, weil dem Anwender diverse Anschlussmöglichkeiten zur Verfügung stehen. Insgesamt können 75 Pins variabel als Ein- oder Ausgang genutzt werden, um externe Peripherie an den FPGA anzubinden.
Für die Fallstudie wurde der Hardware-Inkrementalgeber über das sogenannte Pmod Interface an das Board angeschlossen, bei dem es sich um eine von der Firma Digilent geschaffene 12-polige Hardware-Schnittstelle für Erweiterungskarten (u.a. Audio-CODECS oder LC-Displays) handelt[1]. Zudem bietet das Board eine serielle Schnittstelle an, die dazu genutzt wird die auf dem Mikroprozessor laufenden Programme zu debuggen.

[1] http://www.digilentinc.com/Products/Catalog.cfm?NavPath=2,401&Cat=9
Stand: August 2011

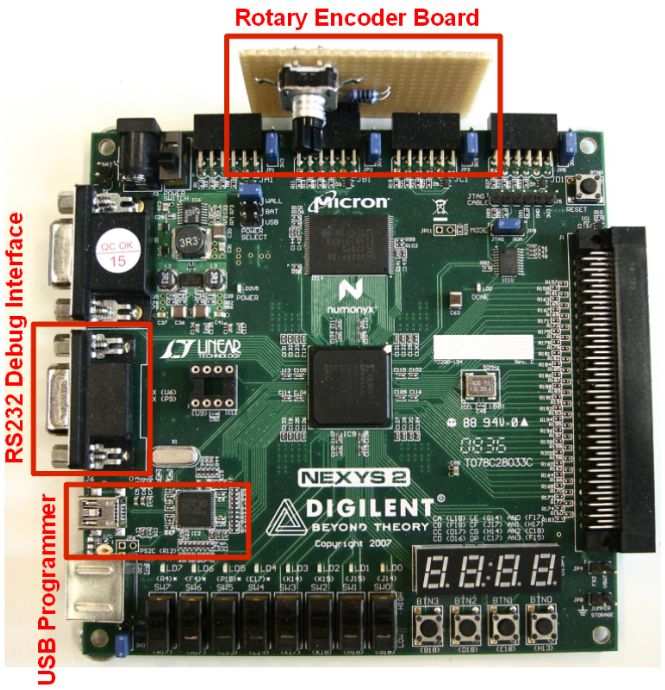

Abbildung 4.2: Das verwendete Digilent Evaluation Board Nexys2 mit dem aufgesteckten Inkrementalgeber-Board

Software Tools

Für die Synthese wurden die Software-Werkzeuge der Firma Xilinx verwendet. Das gesamte Projekt wurde mit ISE in der Version 12.4 verwaltet. Der Entwurf des eingebetteten Systems wurde mit dem Embedded Developer Kit (EDK) erstellt. Für diesen Zweck befinden sich auf der Internetseite der Firma Digilent Dateien, um das Basis-System auf dem Nexys2 Board in das EDK zu laden. In diesem Basis-System sind viele bereits auf dem Board vorhandene Erweiterungen enthalten und können komfortabel im eingebetteten System angesteuert werden, ohne den immensen Konfigurationsaufwand sämtlicher Peripherien manuell durchführen zu müssen.
Die Entwicklung des Quellcodes zur Verifikation des Hardware Abstraction Layers wurde mit dem Software Development Kit (SDK) vorgenommen. Das SDK ist eine auf Eclipse basierende Entwicklungsoberfläche für den Quellcode und das Debuggen des Systems.

Komponenten

Das Xilinx Spartan 3E FPGA bietet, im Gegensatz zu einigen anderen Produktfamilien, keinen in Hardware implementierten Prozessor an, weshalb ein Xilinx Microblaze als Soft-

Prozessor eingesetzt wurde.
Die Anbindung der Hardware-Peripherie erfolgt in der verwendeten EDK-Version über den Peripheral Local Bus (PLB). Der PLB hat den nunmehr obsoleten Onboard Peripheral Bus (OPB) ersetzt. In Verwendung mit einem Microblaze Soft-Prozessor ist der PLB ein 32-Bit breiter Peripherie-Bus, der durch die Firma IBM spezifiziert wurde (siehe [4]).
Die Zugriffe auf die Hardware-Peripherie über den Bus werden auf der Seite des Prozessors über Memory Mapped I/O vorgenommen.

Der IP Core wird unter Verwendung des Intellectual Property Interconnect (IPIC) mit dem Bus verbunden (siehe Abb. 4.3). Die Xilinx Werkzeuge ermöglichen es einen PLB Slave Template zu erzeugen, also eine leere (stub) Komponente, die mit den PLB-Slave-Standardkomponenten an den Prozessor angeschlossen werden kann. Der Abbildung 4.4 kann entnommen werden, wie das Register File und der IP Core in das Xilinx PLB Template integriert werden. Ist das Template erfolgreich mit dem *Create or Import Peripheral...*-Wizard angelegt worden, so erscheint es automatisch im EDK und kann für weitere Projekte verwendet werden. Abschließend muss jedoch nach dem Durchlaufen des Wizard die manuelle Integration des IP Cores Kernels und des Register Files in das Template vorgenommen werden. Als Resultat dieses Integrationsprozesses erhält man den vollständigen IP Core.

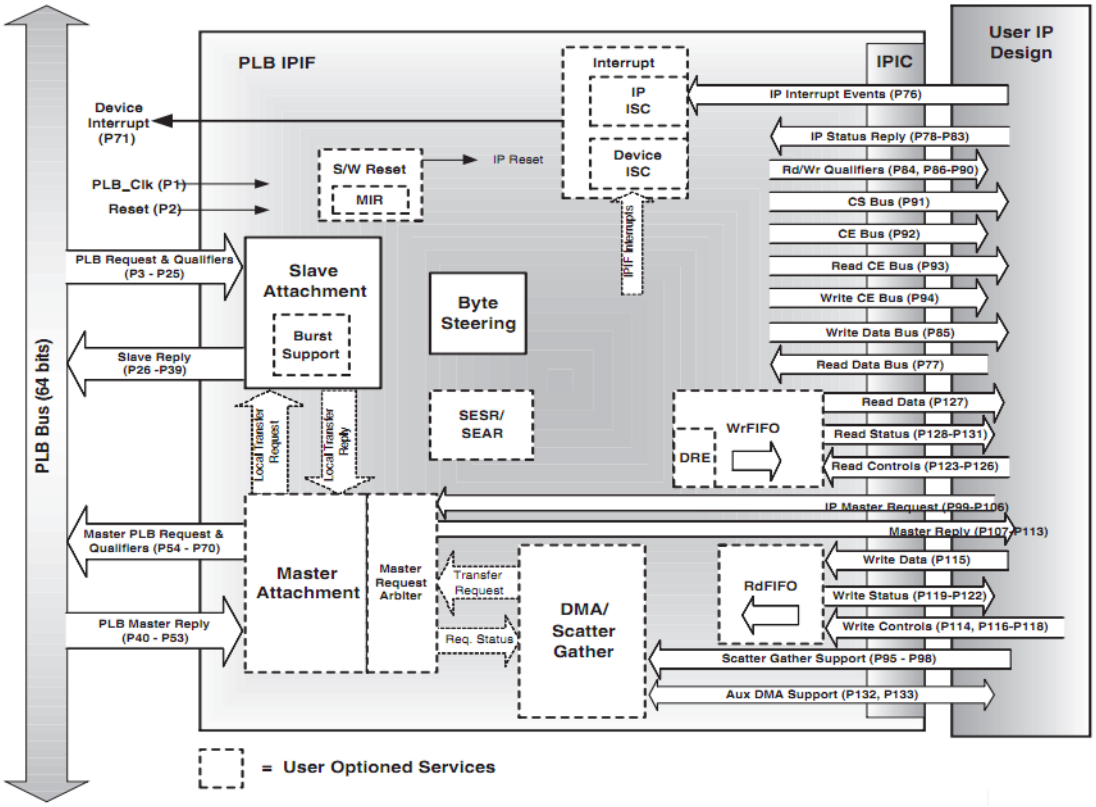

Abbildung 4.3: Anbindung des IP Core an den PLB [16]

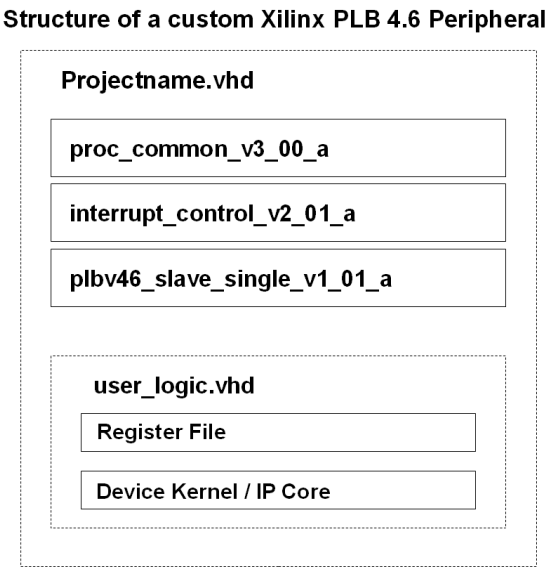

Abbildung 4.4: Integration von benutzerdefinierten IP Cores in das Xilinx PLB-Peripherie-Template

4.4 Modellierung

Die Modellierung des Systems wurde gemäß des vorgestellten Konzepts vorgenommen. In Anlehnung an das MDA-Paradigma wurde ein Modell für die Kommunikation der Schnittstellen und ein weiteres für die Zielplattform erstellt.
Der Entwurf der einzelnen Diagramme wird in ihrer natürlichen Reihenfolge weiter beschrieben, ausgehend von den Anforderungen an das Interface für den Inkrementalgeber.

Das eingebettete System

Das erste Diagramm liefert die Komposition des eingebetteten Systems als Zusammenstellung der Komponenten, die in diesem Gesamtsystem verwendet werden. Das vorliegende ist ein minimalisiertes System, welches für das Ansteuern des Inkrementalgeber entworfen wurde. Die Abbildung 4.5 beschreibt das Blockdefinitionsdiagramm dieses Systems.

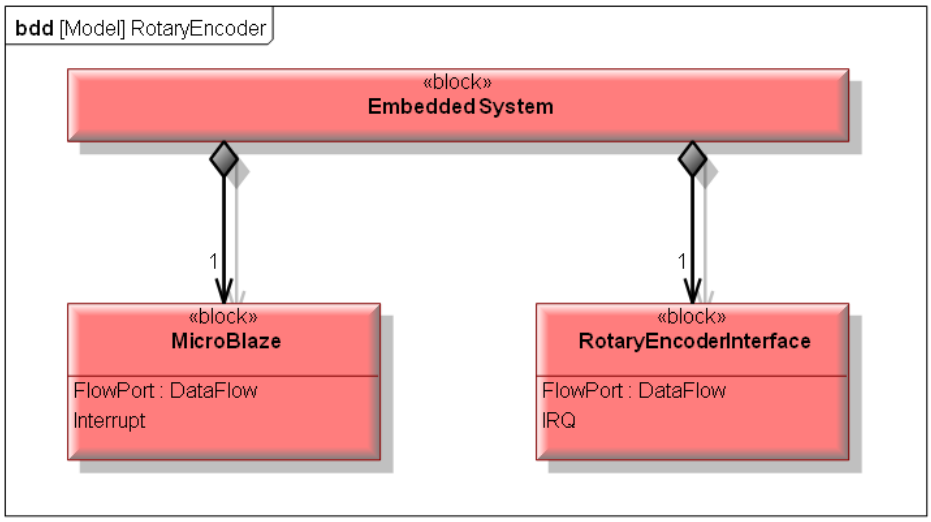

Abbildung 4.5: Komposition des Systems aus IP Core und Mikroprozessor

Verbindung der Teilsysteme

Die Kommunikation der beiden Komponenten kann nun über das interne Blockdiagramm des *Embedded System* (siehe Abb. 4.6) konkretisiert werden. Ports und Konnektoren werden dem Modell hinzugefügt.
Ein Standard Port wird verwendet, um die Interrupt-Signalleitung des Microblaze zur Verfügung zu stellen. Die Spezifikation der Kommunikationsschnittstelle erfolgt über den

Flow Port, der den *Microblaze* Soft-Prozessor und die *RotaryEncoderInterface* Hardware-Komponente beschreibt. Folglich wird die Verbindung der Hardware- und Software-Komponente über dieses interne Blockdiagramm vorgenommen.

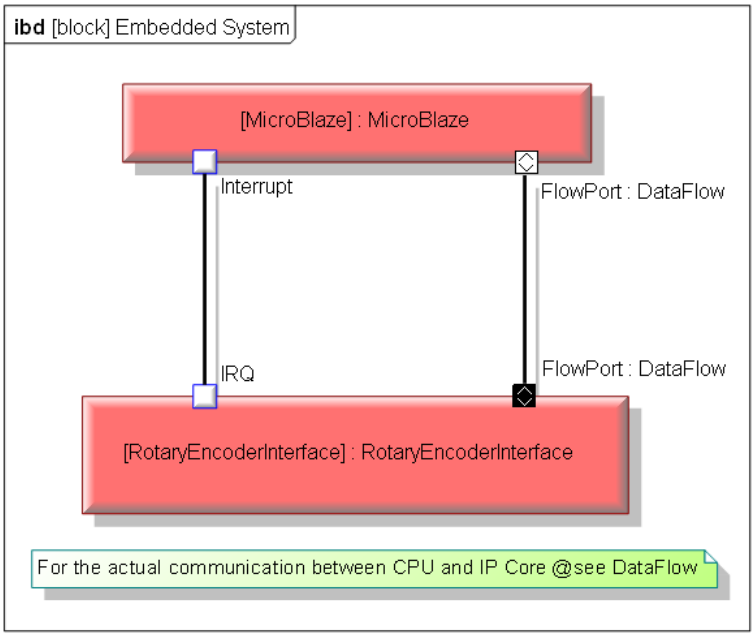

Abbildung 4.6: Die Schnittstellen zwischen Hardware- und Software-Komponenten

Konkretisierung der Kommunikationsobjekte

Darauf folgend wird dem verwendeten Flow Port eine Flow Specification zugewiesen, um die einzelnen Kommunikationsobjekte konkret zu spezifizieren. Die Modellierung erfolgt über ein Blockdefinitionsdiagramm, welches in Abbildung 4.7 zu finden ist. Dort sind zusätzlich die in der Flow Specification verwendeten Aufzählungstypen zur Verdeutlichung der Steuer- und Statusinformationen, die in dieser Schnittstelle verwendet werden, spezifiziert.

Die Zielplattform

Die Zielplattform beschreibt das verwendete Bussystem und die zur Code-Generierung nötigen Details der Software- und Hardware-Architektur. Dazu gehören die Endianess sowohl auf Hardware- als auch auf Software-Seite sowie die Breite des Bussystems in Bit. Das Modell dieser Zielplattform ist in Abbildung 4.8 veranschaulicht. Auf Basis dieser Zielplattform wird das unter der abstrakten Kommunikationsschnittstelle liegende Bussystem konkret.

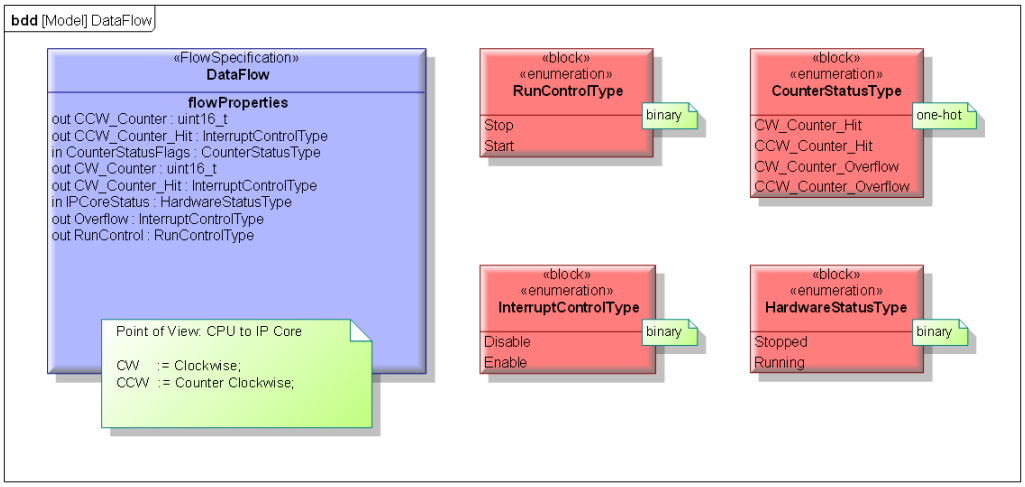

Abbildung 4.7: Übersicht der Schnittstellenspezifikation (DataFlow), nebenstehend annotiert die verwendeten Aufzählungstypen

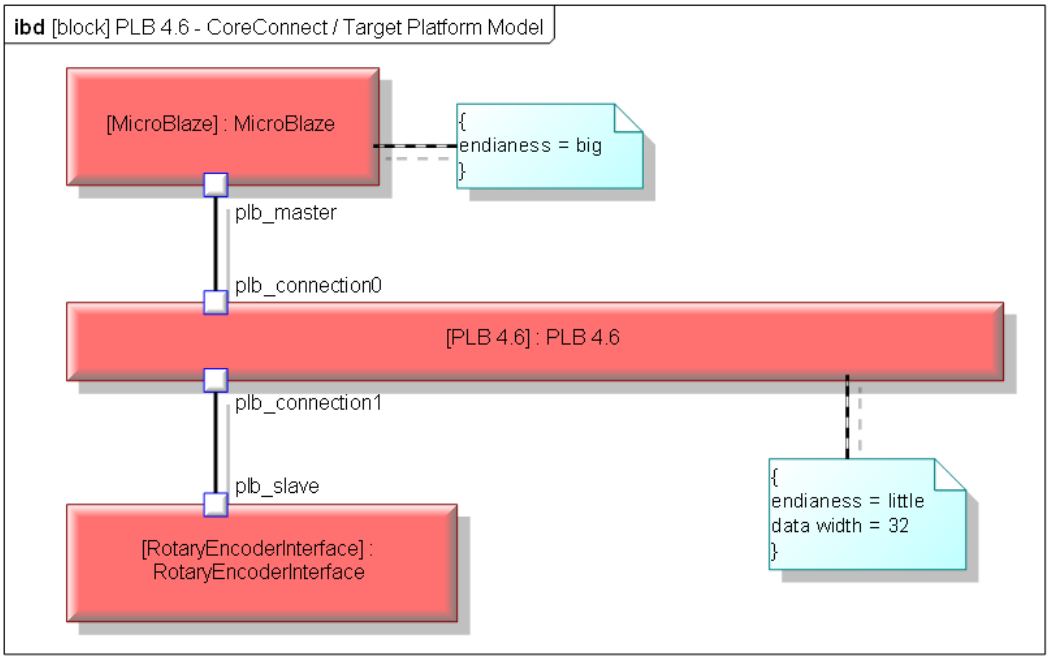

Abbildung 4.8: Das Modell der Zielplattform

4.5 Transformation

4.5.1 Transformationsregeln

Zur Realisierung einer Transformation aus dem SysML-Modell ist es unumgänglich, dass Transformationsregeln definiert und in der Modellierungsphase eingehalten werden. Einige Regeln, wie z.B. Typkonventionen, können angepasst werden, wenn der implementierte Code-Generator dementsprechend verändert wird: Andere, wie z.B. die Namenskonventionen, sind unveränderbar, da diese ein elementares Konzept des Werkzeugs darstellen.
Nachfolgend werden geltende Konventionen und deren Entstehung aufgezeigt.

Namenskonventionen

Für die Modellierung der einzelnen Blöcke, die Typen definieren oder später als Flow Property eingesetzt werden, müssen Namenskonventionen für die Realisierung einer konsistenten Code-Generierung erstellt werden. Da die Zielsprache in dieser Arbeit *C* ist, werden die ANSI-C Namenskonventionen verwendet. Die Namen müssen so gewählt werden, dass jederzeit eine Bijektivität zwischen Modell und Code-Generaten besteht.
Folglich müssen die Namen eineindeutig sein und nicht aus dem Vokabular der Sprache *C* stammen.

Typkonventionen

Datentypen müssen aus dem ANSI-C99 *Exact-width integer types* Set (int$N$_t oder uint$N$_t | $N \in [8, 16, 32, 64]$) gewählt werden, die in der Datei *stdint.h* definiert und auf den meisten modernen Mikroprozessoren bereits verfügbar sind.
Der Grund für diese Restriktion ist, dass die Bitbreite der normalen Integer-Typen je nach Zielarchitektur variieren kann. Zusätzlich kann als Datentyp, der nach IEEE-754 standardisierte Gleitkommazahlentyp *float* verwendet werden. Kontroll- und Statustypen werden für das in dieser Arbeit verwendete Konzept immer als Aufzählungstypen modelliert und automatisch auf entsprechende primitive Datentypen gemappt.

Größenkonventionen

Die verschiedenen Bussysteme zur Anbindung von Hardware-Peripherie besitzen nur eine endliche Anzahl an Adressierungsbits, welche implizit die maximale Größe des Register File bestimmen.

Das vorliegende PLB-System kann mit maximal 4096 Register pro PLB-Peripherie ausgestattet werden, da dies im Xilinx Wizard für die Erstellung zusätzlicher Hardware-Peripherie auf diesen Wert begrenzt wird.
Bei der Modellierung sollte deshalb, unter Verwendung des Code-Generators, auch während der Entwicklung schon einmal überprüft werden, ob dieses Limit noch nicht erreicht wird. Maximal können so 16 MByte adressiert werden, die für die meisten Projekte ausreichend sind.

Modellierung des Standpunktes

Eine einheitliche Sicht auf das System zu definieren erwies sich als sinnvoll. Über die Modellierung des Standpunktes wird angegeben, in welcher Richtung der Informationsfluss interpretiert wird. Hat z.B. eine Flow Specification ein Element der Richtung *in*, ist nicht sofort ersichtlich, an welcher Seite des Systems der nicht konjugierte Flow Port angebracht werden soll.
Daher wird empfohlen den Standpunkt der Kommunikation mit der Perspektive so zu definieren, dass eine Sicht vom Prozessor zum IP Core besteht.
Der Mikroprozessor stellt in den meisten Fällen das komplexere System im Vergleich zur Hardware-Peripherie dar und bietet sich deshalb als Standpunkt deutlich an, da der Blick vom komplexeren auf das weniger komplexe System erfolgen sollte. Der in Kapitel 4.6 vorgestellte Generator baut auf dieser Sicht auf, ist jedoch flexibel in Bezug auf die Modellierung, da eine eindeutige Angabe der Kommunikationsrichtung im eigenen Eingabeformat für jedes Objekt definiert werden muss. Die Definition der Sicht ist für den implementierten Code-Generator nicht obligatorisch, da die absolute Flussrichtung folglich gekennzeichnet wird.

Es empfiehlt sich trotzdem eine einheitliche Sicht zur Gewährleistung der Konsistenz des Modells von Anfang an zu definieren und auch beizubehalten.

4.5.2 Modell-zu-Modell-Transformation

Zur Entkopplung der formalen SysML-Spezifikation vom Modellierungswerkzeug und zur weiteren Verwendung des Modells muss zunächst eine Modell-zu-Modell-Transformation auf das SysML-Modell angewendet werden, die die notwendigen Informationen des Modells extrahiert und zusammenfasst.
Dabei ist es von größter Bedeutung, dass das Quellmodell nicht verändert werden darf.
In dieser Arbeit wird eine manuelle Modell-zu-Text-Transformation vorgenommen, um aus dem SysML-Modell ein definiertes Eingangsformat für den im Folgenden vorgestellten Code-Generator zu erzeugen.

Auch eine automatisierte Transformation wäre unter Verwendung von XSLT oder Parser-Generatoren durchaus vorstellbar, da das Ausgangsformat der Modellierungswerkzeuge ein XML-Format liefert. Da diese vom Modellierungswerkzeug abhängig wäre und keinen nennenswerten Beitrag für diese Arbeit liefern würde, wurde diese Art der Transformation in dieser Arbeit nicht realisiert.
Ein eigenes Eingabeformat, welches durch den Menschen lesbar und editierbar ist, stellte deshalb die beste Lösung dar das SysML-Modell unmittelbar und effizient zu verwenden (siehe Abschnitt 4.6.2).

4.6 Der Code-Generator: SyCoGen

Der für diese Arbeit entworfene SysML-Code-Generatorprototyp (SyCoGen) arbeitet, wie viele andere Generatoren auch, sequentiell. Nachdem die formale Beschreibung eingelesen (geparsed) wurde, wird sukzessiv die interne Datenstruktur im Speicher aufgebaut. Die interne Datenstruktur repräsentiert dabei alle Deklarationen, Allokationen und Mappings.
Im letzten Schritt werden die verschiedenen Generatoren aufgerufen, um den HAL, das Register File und zusätzliche Informationen für den Benutzer zu erzeugen. Zum Erreichen einer möglichst Plattform unabhängigen Lösung wurde der SyCoGen vollständig in Java implementiert. Während der Konzeptionsphase des Generators wurde besonderer Wert auf die Entwicklung einer möglichst flexiblen und erweiterbaren Lösung gelegt.

4.6.1 Code-Generierung mit SyCoGen

Die Code-Generierung mit dem SyCoGen arbeitet auf einer Toolchain-Basis, die die verschiedenen zur Generierung nötigen Schritte in mehrere Phasen unterteilt. Diese werden, wie in Abbildung 4.9 ersichtlich, nacheinander abgearbeitet.
Den ersten Schritt für die Code-Generierung stellt die Modell-zu-Text-Transformation aus der in SysML modellierten Spezifikation dar. Als Produkt dieses Schrittes erhält man das Inputfile für den Generator. Anschließend wird der Generator mit dem Inputfile als Argument aufgerufen. Als Produkte (Generate) dieses Vorganges erhält man den HAL, in Form einer C-Header Datei, und ein Register File, welches als VHDL-Beschreibung generiert wird.
Die Datei für den HAL wird über ein Inkludieren in das Software-Projekt eingebunden.
Hingegen muss das Register File händisch in das Xilinx PLB Template integriert werden. Für diesen Zweck wird ebenfalls manuell vom Anwender unter Verwendung eines Wizards das Xilinx PLB Slave Template im Embedded Developer Kit (EDK) erzeugt. Zur Integration des Register Files in dieses Template stellt der SyCoGen für diesen Zweck den sogenannten Infogenerator zur Verfügung. Dieser unterstützt den Anwender, indem die Bit-Vektor-Positionen der einzelnen Objekte des Register Files in textueller Form ausgegeben werden.

Die Abbildung 4.10 stellt graphisch den Syntheseablauf unter Verwendung des SyCoGen und der Xilinx Toolchain dar. Im letzten Schritt dieser Synthese findet schließlich die Systemsynthese auf der Zielplattform (dem FPGA) statt.

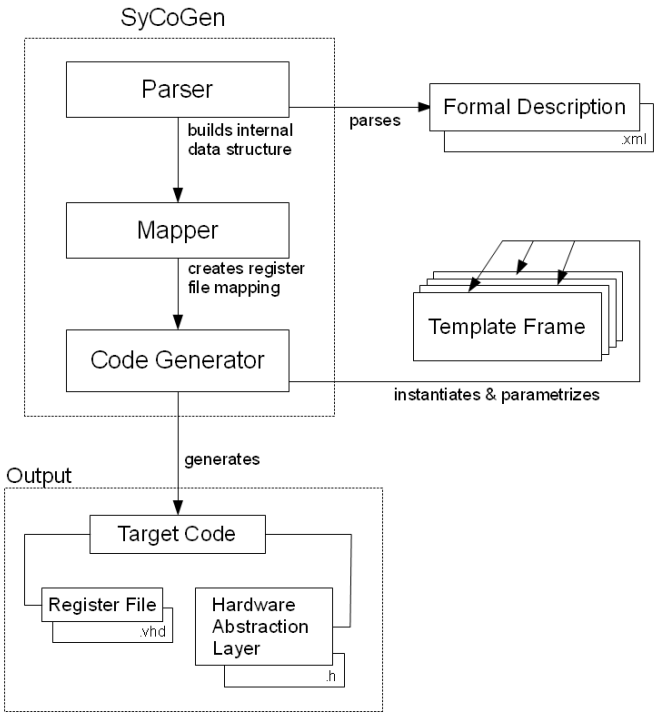

Abbildung 4.9: Code-Generierung mit SyCoGen

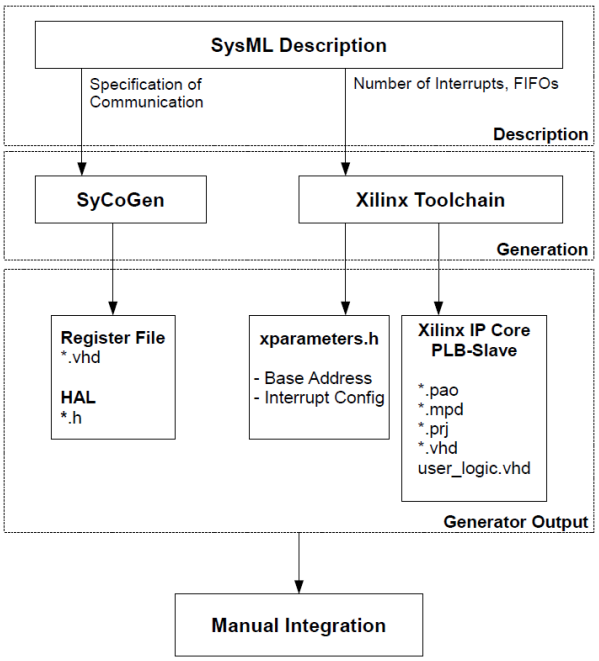

Abbildung 4.10: Synthese mit SyCoGen

4.6.2 Eingabeformat

Obwohl das Ausgabeformat der Modellierungswerkzeuge von der OMG über das sogenannte *XML Metadata Interchange*[2]-Format (XMI-Format) spezifiziert ist, unterscheidet sich dieses jedoch von Hersteller zu Hersteller immens. Ein XML-Format wurde eigens entworfen, um ein für den Menschen lesbares Dokument bereitzustellen. Dieses Format ist auf die Beschreibung der Schnittstelle zwischen Hardware und Software reduziert.

Das Eingabeformat lässt sich in drei Schichten unterteilen (siehe Abb. 4.11), die im Nachfolgenden weiter beschrieben werden.

Konfiguration: *configuration*

In der Konfigurationsschicht wird die Endianess des Mikroprozessors und die des Bussystems definiert. Die von den FPGA-Herstellern zur Verfügung gestellten Bus-Slave-Module für benutzerdefinierte IP Cores bieten häufig direkt eine Schnittstelle in Little-Endian-Darstellung für ihre Hardware-Peripherie an. Zur Unterstützung des sehr allgemein gehaltenen Konzepts muss jedoch diese Endianess trotzdem angegeben werden, damit auch andere Systeme verwendet werden können.
Zusätzlich muss noch ein Projektname angegeben werden, der in der Generierungsphase als Präfix für Ausgangsdateien und Makros genutzt wird.

Deklaration: *declaration*

Die im SysML-Modell verwendeten Datentypen (sowohl die primitiven als auch die Aufzählungstypen) werden in diesem Abschnitt deklariert.
Zur Vermeidung von Kollisionen müssen alle definierten Datentypen einen eineindeutigen Bezeichner besitzen.
Zusätzlich muss für primitive Datentypen die Größe in Bit angegeben werden, damit der Generator ein Mapping durchführen kann. Für Aufzählungstypen gilt dies nicht. Das Encoding muss jedoch aus demselben Grunde gewählt werden, damit die Größe daraus berechnet wird und somit implizit gegeben ist.

Allokation: *allocation*

Die Allokationsschicht stellt die textuelle Representation der bereits erwähnten SysML Flow Specification dar. Zur Verdeutlichung, dass die zu allokierenden Objekte, Speicherobjekte

[2] http://www.omg.org/spec/XMI/ Stand: August 2011

sind, die sich in Registern befinden, werden diese Objekte hier als Memoryobject bezeichnet und nicht mehr als Flow Property, wie es im SysML-Modell der Fall wäre. Ein Memoryobject setzt sich, genauso wie die Flow Property, aus einem Namen (*name*), einem Typ (*type*) und der Flussrichtung (*direction*) zusammen.

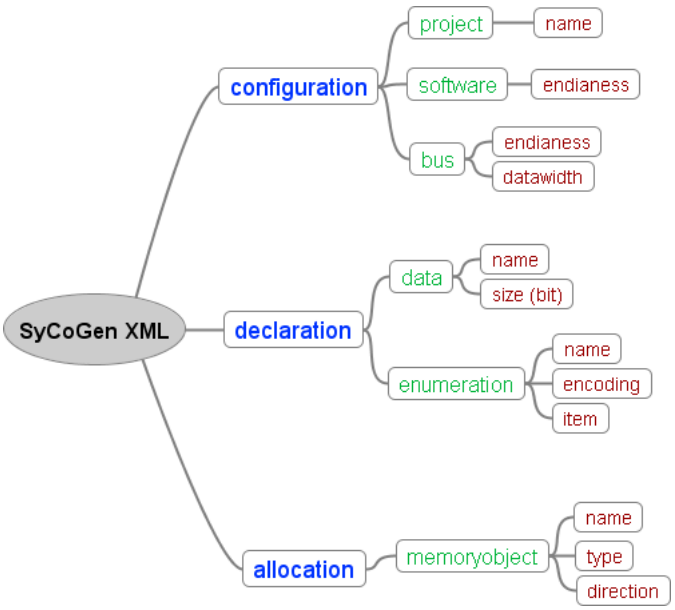

Abbildung 4.11: Aufbau des SyCoGen-XML-Formats

Das folgende Listing 4.1 zeigt das durch die manuelle Modell-zu-Text-Transformation entstandene XML Inputfile für den SyCoGen. Die als Kommentar annotierten Informationen bezeichnen den Ursprung des jeweiligen Abschnittes. Hierdurch ist es möglich direkt auf die genutzten Modelle zu schließen und die Transformation nachzuvollziehen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- @author     Fabian Zahn 2011 -->
3  <!-- @title      RotaryEncoder -->
4
5  <SyCoGen version="alpha">
6      <!-- bdd: [Model]RotaryEncoder -->
7      <configuration>
8          <project     name="RotaryEncoder" />
9          <software    endianess="big" />
10         <bus         endianess="little" datawidth="32"/>
11     </configuration>
12
13     <!-- package: MyDataTypes -->
14     <declaration>
15         <!-- "MyDataTypes" -> "C99" -->
16         <data name="uint8_t"     size ="8"/>
17         <data name="uint16_t"    size ="16"/>
```

```
18          <data name="uint32_t"   size ="32"/>
19          <data name="uint64_t"   size ="64"/>
20          <data name="int8_t"     size ="8"/>
21          <data name="int16_t"    size ="16"/>
22          <data name="int32_t"    size ="32"/>
23          <data name="int64_t"    size ="64"/>
24
25          <!-- "MyDataTypes" -> "IEEE 754" -->
26          <data name="float" size="32"/>
27          <data name="double" size="64"/>
28
29          <enumeration name="RunControlType" encoding="binary">
30              <item name="Stop"/>
31              <item name="Start"/>
32          </enumeration>
33
34          <enumeration name="InterruptControlType" encoding="binary">
35              <item name="Disable"/>
36              <item name="Enable"/>
37          </enumeration>
38
39          <enumeration name="CounterStatusType" encoding="one-hot">
40              <item name="CW_Counter_Hit" />
41              <item name="CCW_Counter_Hit" />
42              <item name="CW_Counter_Overflow" />
43              <item name="CCW_Counter_Overflow" />
44          </enumeration>
45
46          <enumeration name="HardwareStatusType" encoding="binary">
47              <item name="Stopped"/>
48              <item name="Running"/>
49          </enumeration>
50      </declaration>
51
52      <!-- bdd: [Model]DataFlow; ibd: [block]Embedded System -->
53      <allocation>
54          <!-- in -->
55          <memoryobject name="CounterStatusFlags" type="CounterStatusType"    direction="hw2sw" />
56          <memoryobject name="IPCoreStatus"       type="HardwareStatusType"   direction="hw2sw" />
57
58          <!-- out -->
59          <memoryobject name="CW_Counter"         type="uint16_t"             direction="sw2hw" />
60          <memoryobject name="CW_Counter_Hit"     type="InterruptControlType" direction="sw2hw" />
61          <memoryobject name="CCW_Counter"        type="uint16_t"             direction="sw2hw" />
62          <memoryobject name="CCW_Counter_Hit"    type="InterruptControlType" direction="sw2hw" />
63          <memoryobject name="Overflow"           type="InterruptControlType" direction="sw2hw" />
64          <memoryobject name="RunControl"         type="RunControlType"       direction="sw2hw" />
65      </allocation>
66  </SyCoGen>
```

Listing 4.1: SyCoGen Inputfile: RotaryEncoder.xml

4.6.3 Mapping

Als Mapping bezeichnet man den Prozess der Abbildung von Objekten auf das Register File. Gegebene Allokationen von Datentypen werden den verschiedenen Registern zugewiesen. Das Produkt des Mapping-Prozesses liefert eine in sich konsistente Verteilung der einzelnen Objekte im Register File.

Die Allokations- und Deklarationsbeschreibungen werden während des Mapping-Vorganges genutzt, um die sogenannten Memoryobjects zusammenzusetzen. Die Memoryobjects stellen konkrete Objekte in den Registern dar. Nur wenn eine konsistente Beschreibung der Kommunikation im SyCoGen-XML-Format gegeben ist, kann ein gültiges Memoryobject unter Verwendung von Allokations- und Deklarationsbeschreibung entstehen (siehe Abb. 4.12).

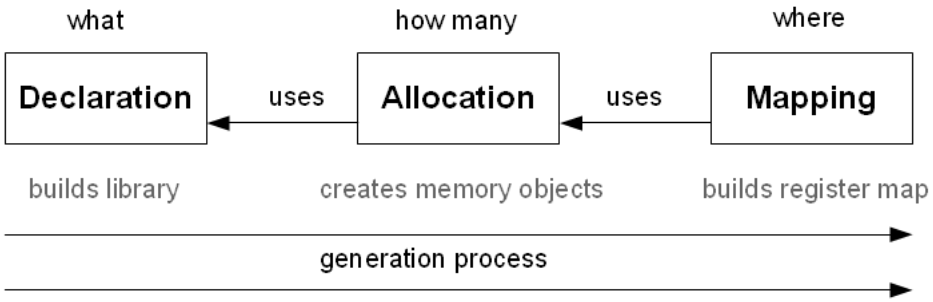

Abbildung 4.12: Der Mapping-Vorgang von der Spezifikation zum konkreten Memoryobject

Trennung der Daten

Für den Mapping-Vorgang werden die einzelnen Objekte vorsortiert, damit eine lokale Trennung von Steuerungs-, Kontroll-, Eingangs- und Ausgangsinformationen erzeugt werden kann (siehe Abbildung 4.13).
Die Sortierung ist zur Definition verschiedener Operationen auf die Register notwendig. Leseoperationen dürfen auf alle Register durchgeführt werden, Schreiboperationen dürfen von der Software-Seite jedoch nur in den Steuerungs- und Ausgangsdatenpartitionen des Register Files durchgeführt werden. Auf der Hardware-Seite dürfen demnach auch nur Schreiboperationen in den Status- und Dateneingangsteilen des Register Files vorgenommen werden.

Nach der Sortierung werden die einzelnen Objekte den verschiedenen Registern, welche dieselbe Breite wie die Busbreite besitzen, zugeordnet. Die Implementierung des SyCoGen-Werkzeugs erfolgte dabei diesen Regeln und baut auf dieser Sicht auf das System auf. Eine

Fragmentierung einzelner Objekte wird derzeit nicht unterstützt. Sie wäre auch nur für Datenobjekte möglich. Versuchte man ein Objekt, welches als Steuerungs- oder Statusinformation von der Peripherie verwendet wird, in kleinere Objekte zu zerlegen, so wären die Lese- und Schreiboperationen, die über den Bus ausgeführt werden, nicht mehr atomar. Möglicherweise ergäben sich Dateninkonsistenzen, die zu fehlerhaftem Verhalten auf beiden Seiten der Kommunikation führen könnten. Beispielsweise wird hierfür eine Statusinformation in 2-Bit kodiert, was vier verschiedene Nachrichten ermöglicht. Durch die zwei zeitversetzten Lesevorgänge, die benötigt werden, um den Status auszulesen, könnten eventuell im Empfänger falsche Daten entstehen, sofern sich der Status zwischen diesen beiden Lesevorgängen geändert hat.
Aus diesem Grund ist es auch nicht erlaubt in der Modellierung von Status- und Steuerinformationen Objekte zu erstellen, die größer als die Busbreite bzw. damit auch als die Registerbreite sind.

Mapping-Algorithmus

Für das Mapping wird ein lokal optimierender Algorithmus aus der Klasse der Greedy-Algorithmen verwendet, der sogenannte First-Fit-Decreasing-Algorithmus (FFD). Durch die zum aktuellen Zeitpunkt beste lokale Erweiterung der Objekt-Map wird das Mapping sukzessive durchgeführt. Für diesen Zweck werden die Memoryobjects im ersten Schritt absteigend nach ihrer Größe in Bit sortiert und anschließend in die entsprechenden Register gesetzt.

Demnach wird anfangs das größte Objekt in ein Register gesetzt, dann das zweitgrößte. Sollte dies jedoch nicht passen, so wird dieses wiederum in ein neues Register gesetzt. Das drittgrößte Objekt wird danach versucht in das erste, nicht volle Register zu setzen, sofern dies nicht möglich ist, in das zweite Register und diesem Prinzip folgend solange bis ein freier Platz gefunden wird oder ein neues Register erstellt wurde, um das Objekt aufzunehmen.
Iterativ wird die jeweils lokal beste Lösung gefunden, um die Anzahl von Registern minimal zu halten. Allerdings liefert dieser Algorithmus nicht immer eine optimale Lösung in Bezug auf das gesamte System.
Über die vorhandenen Gleichungen (Gleichung 4.1 und 4.2) kann der Registerverbrauch im Vorfeld für den Worst- bzw. Best-Case berechnet werden. $\mathrm{OPT}(L)$ liefert die optimale Anzahl an Registern für eine gegebene Eingangsfolge L.

Für alle Eingangsfolgen L gilt Gleichung 4.1, außerdem gibt es Eingangsfolgen L für die Gleichung 4.2 gilt.

$$\mathrm{FFD}(L) \leq \frac{11}{9} \cdot \mathrm{OPT}(L) + 1, \quad \forall L \tag{4.1}$$

$$\mathrm{FFD}(L) = \mathrm{OPT}(L), \quad \exists L \tag{4.2}$$

Der Beweis für Gleichung 4.1 kann in [17] eingesehen werden.
Die Gleichung 4.2 kann hingegen sehr einfach bewiesen werden:
$\mathrm{FDD}(L)$ ist äquivalent zu $\mathrm{OPT}(L)$, für Eingangsfolgen, die alle Register vollständig füllen und damit zu einhundert Prozent belegt sind. Als fiktives Beispiel sei ein 32-Bit breites Register genannt, welches mit zwei Datentypen *uint_16* belegt wird. Das Register wird mit dem verwendeten Algorithmus optimal mit zwei 16-Bit breiten Datentypen belegt.
Demnach gilt in Bezug auf diese Eingangsfolge L: $\mathrm{FDD}(L) = \mathrm{OPT}(L)$.

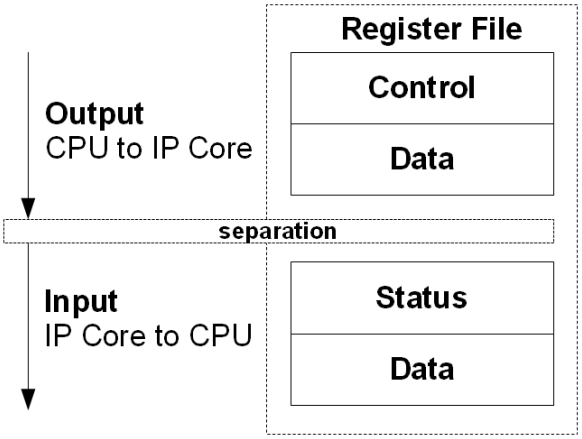

Abbildung 4.13: Trennung der verschiedenen Kommunikationsobjekte

4.6.4 Frame Processing

Die Implementierung des Code-Generators basiert auf dem Prinzip des Frame Processing. Beim Frame Processing werden kleine Code-Fragmente, die sogenannten Frames, eingelesen, ausgewertet und zu einem Gesamten zusammengefügt[13]. Frame Processing wird sehr häufig verwendet und ist ein sehr flexibler Ansatz der template-basierten Code-Generierung. Die Lernkurve für den Entwickler von Templates ist relativ flach, da die Komplexität der einzelnen Templates (Frames genannt) begrenzt ist. Im Laufe der Entwicklung des Code-Generators wurde jedoch deutlich, dass das Frame Processing für Sprachen wie VHDL an die Grenzen des Überschaubaren stößt.
Eine vollständige VHDL-Beschreibung eines Register Files mit getrennten Prozessen für Lese- und Schreiboperationen auf die Register, deren räumliche Trennung und die Adressdekodierung des Busses sorgen dafür, dass das Code-Generat aus sehr vielen einzelnen Frames zusammengesetzt werden muss. Der Entwickler, der nun versucht die Templates zu modifizieren, muss jederzeit um die Folgen in den anderen Templates wissen.

Für die Generierung des HAL war die Lösung mittels Frame Processing eine sehr gute Wahl. Der HAL stellt eine Komposition aus Getter-, Setter-, Typedefinitons- und Wertdefinitions-Makros dar, die jeweils von einander unabhängig sind. Lediglich der Datentyp der einzelnen Komponenten ist gemeinsam, wird jedoch mittels Code-Generierung erzeugt und damit wiederum unabhängig. Selbst für unerfahrene Entwickler ist die Anpassung der HAL-Quellcode-Templates dadurch unkompliziert möglich. Beispielsweise sei der Wechsel von Memory Mapped I/O auf einen dedizierten Bustreiber genannt, dies erforderte nur die Anpassung von zwei Zeilen in jedem Template.

4.6.5 Code-Generate

In seiner aktuellen Implementierung liefert der SyCoGen zwei für die Realisierung nutzbare Generate, wovon eines das Register File darstellt und das weitere den Hardware Abstraction Layer.
Zusätzlich werden Informationen ausgegeben, die für die Integration des Register Files in das Xilinx PLB Template benötigt werden. Diese sind jedoch eher formeller Natur und stellen kein synthetisierbares bzw. kompilierbares Generat dar. Nachfolgend werden deshalb die Generate beschrieben, die tatsächlich als Teil des Gesamtsystems in das Projekt integriert werden.

Register File

Das Register File wird in einer VHDL-Datei beschrieben und wird, wie im Gesamtkonzept festgehalten, als Komposition aus Flip-Flops gebildetet. Ein Register wird für die verwendete Plattform aus 32 Flip-Flops zusammengesetzt und als ein VHDL-Prozess modelliert. Der Adressdecoder für Schreiboperationen, die vom Bus auf das Register File durchgeführt werden, wird aus praktischen Gründen als Chip-Enable-Signal der einzelnen Register implementiert. Folglich wird die Code-Generierung eines Registers durch diese Art der Implementierung einfacher, da dieser demzufolge keinen eigenen Prozess mehr erfordert.
Der Lesedecoder, der die Lesezugriffe der Software-Seite auf das Register File umsetzt, wird hingegen als getrennter Prozess beschrieben.

- Decoder für Schreiboperationen
- Decoder für Leseoperationen
- Schnittstelle für Byte-Enable-Signal
- N-Bit breite Register

Die Abbildung 4.14 zeigt das von dem Xilinx ISE-Werkzeug generierte Blockschaltbild des Register Files. Wie man erkennen kann, wurden drei Register für die Speicherung der Kommunikationsobjekte generiert. Die *bus2ip_data* bzw. *ip2bus_data* Schnittstellen werden genutzt um Daten vom Bus in das Register File zu schreiben bzw. Daten aus dem Register File über den Bus auszulesen.

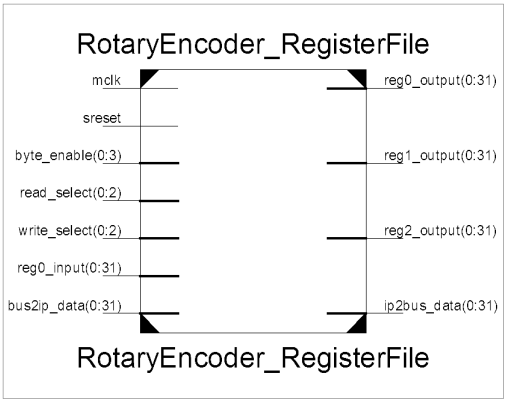

Abbildung 4.14: Schematische Darstellung des Register Files als Baustein digitaler Logik

Hardware Abstraction Layer

Die Hardware-Abstraktionsschicht setzt sich vollständig zusammen aus Makros, Preprozessor-Anweisungen und Typdefinitionen. Für die verschiedenen Partitionen des Register Files werden individuell für die Nutzung der Kommunikationsobjekte nötige Bestandteile generiert. Die Tabelle 4.1 zeigt die einzelnen Elemente, aus denen sich der HAL für die verschiedenen Objekttypen zusammensetzt.

	Getter Makro	**Setter Makro**	**Wertdefinition**	**Typdefinition**
Control	x	x	x	x
Data Out	x	x		
Status	x		x	x
Data In	x			

Tabelle 4.1: HAL Generate

Das Listing 4.2 zeigt den generierten Quellcode für Kontrolltyp *RunControl*, der genutzt wird, um den IP Core zu starten oder zu stoppen.

```
1  // typedefinition (mapped to the smallest available datatype)
2  typedef uint8_t RunControl;
3
4  // valuedefinition (binary encoding)
5  #define RotaryEncoder_RunControl_Stop   0x00000000
6  #define RotaryEncoder_RunControl_Start  0x00000001
7
8  // getter-macro (returns a value read -> element of valuedefinition)
9  #define RotaryEncoder_getRunControl(void) \
10     ( (RunControl) \
11         ((PLB_READ_32BIT \
12             ( \
13                 ( \
14                     (RotaryEncoder_BASEADRESS)+(1*4) \
15                 ) \
16             ) \
17             RIGHT_SHIFT_BY (0)) & 0x00000001) \
18     )
19
20 // setter-macro (sets a given value -> element of valuedefinition)
21 #define RotaryEncoder_setRunControl(value) \
22     PLB_WRITE_32BIT \
23     ( \
24         (RotaryEncoder_BASEADRESS)+(1*4), \
25         (PLB_READ_32BIT \
26         ( \
27             ( \
28                 (RotaryEncoder_BASEADRESS)+(1*4) \
29             ) \
30         )&= (uint32_t)~(0x00000001 LEFT_SHIFT_BY (0)))| \
31             ( \
32                 (value) LEFT_SHIFT_BY (0)     \
33             ) \
34     )
35 \label{code:bsp-hal-code}
```

Listing 4.2: Exemplarisches Code-Generat eines Control-Typs

4.7 Verifikation der Realisierung

Zur strukturellen Verifikation der Generate wurde überprüft, ob sich der HAL fehlerfrei bzw. frei von Compiler Warnings kompilieren lässt. Das Register File wurde separat durch eine VHDL-Synthese auf die korrekte Verwendung der sprachlichen Syntax und die Synthetisierbarkeit überprüft.

Ebenfalls wurde der für diese Fallstudie entworfene Kernel des IP Core auf diese Art und Weise verifiziert und zusätzlich mit einer Testbench auf die Funktionalität der signifikantesten Eigenschaften getestet. Ein Test, der sämtliche Testszenarien abdeckt, wurde nicht vorgenommen, da dies kein Bestandteil dieser Arbeit ist, die lediglich die Richtigkeit der Schnittstelle garantieren soll.

Für die funktionelle Evaluation der Schnittstelle wurden mehrere Tests vorgenommen. Die verschiedenen Datentypen des verwendeten Interfaces wurden mittels der generierten HAL-Makros auf der Software-Seite des Systems überprüft, indem definierte Werte in das Register File geschrieben und anschließend zurückgelesen wurden.
Zudem wurden nach den Schreiboperationen Dumps des Register Files ausgelesen und manuell auf ihre Richtigkeit überprüft. Abschließend wurde ein Funktionstest des Gesamtsystems durchgeführt, indem der IP Core für einen exemplarischen (fiktiven) Aufbau konfiguriert wurde. Durch einfache Tests, wie das Erhalten und Auswerten eines Interrupt-Signals nach x Umdrehungen des Inkrementalencoders in Richtung y oder das Deaktivieren des IP Cores nach einem Interrupt, wurde das System verifiziert.

Eine Prüfung der Generate des Code-Generators erfolgte durch das Variieren von Inputfiles. Hierzu sind verschiedene Modifikationen an diesem vorgenommen worden. Mit Hilfe einer anschliessenden manuellen Auswertung über Dateidifferenzen (diff) wurde die Richtigkeit der Generate validiert. An dieser Stelle sei jedoch noch einmal hervorgehoben, dass es sich beim SyCoGen um ein Proof of Concept handelt, das nicht ohne weitere Tests für den Produktiveinsatz verwendet werden sollte.

4.8 Validierung des Konzepts

Der Beweis wurde erbracht, dass aus einer formalen SysML-Spezifikation einer Hardware/Software-Schnittstelle diese automatisiert erzeugt werden kann. Abschließend sollte jedoch noch einmal kritisch betrachtet werden, ob das gewählte Konzept optimal ist und damit sämtliche Bedingungen erfüllt sind. Deshalb wird in den folgenden Absätzen diese Reflexion vorgenommen, die dazu dient Vor- und Nachteile des genutzten Konzepts herauszuarbeiten.

Wie sich zeigte, eignet sich die SysML sehr gut dafür, um mit ihren Flow Ports Interfaces im Allgemeinen und damit auch Hardware/Software Interfaces zu beschreiben. Die vorliegende Flow Specification betrachtend fällt sofort auf, dass eine totale Übersicht für den Außenstehenden nicht mehr gegeben ist.
Für sehr komplexe Schnittstellen empfiehlt es sich, die einzelnen Bereiche der Kommunikation (Steuerungs-/Statusinformationen und Daten) in einzelne Flow Ports mit jeweils eigenen Flow Specifications zu modellieren.

Darüber hinaus wird von den Entwicklern ein Bruch mit traditionellen Methoden im Umgang mit Register Files gefordert. Die Set- bzw. Clear-Mechaniken der Register, wie sie von einigen Mikrocontrollern bekannt sind, werden für diesen Ansatz durch unidirektionale Register ersetzt, die diese Operationen nicht anbieten. Eine Clear-Operation auf ein Status Flag ist so beispielsweise nicht mehr möglich.
Die kollisionsfreie Entwicklung stellt jedoch immer einen Anspruch an den Entwickler dar. Das vorliegende Konzept ist möglicherweise ein guter Schritt in Richtung der strikten Trennung von Zuständigkeiten das Register File betreffend.

Hinsichtlich der Code-Generierung erwies sich das Konzept als sehr gut nutzbar. Auf wichtige Eigenschaften, wie z.B. die Eineindeutigkeit und die Trennung von Kommunikationsinformationen, wurde seit Beginn dieser Arbeit großer Wert gelegt.

5 Zusammenfassung und Ausblick

Zusammenfassung

Diese Arbeit zeigt, dass unter Verwendung der Modellierungssprache SysML die Modellierung von Kommunikationskanälen zwischen Hardware/Software-Endpunkten möglich ist. Auf abstrakter Ebene entstandene Modelle der Kommunikationskanäle können, unter Verwendung des SyCoGen-Werkzeugs und bestimmten Konventionen in Bezug auf die Modellierung, dazu genutzt werden sowohl die Hardware-Abstraktionsschicht als auch das Register File zu erzeugen.
Durch Integration dieser Code-Generate in die entsprechenden Xilinx Werkzeuge ist die Automatisierung eines großen und fehleranfälligen Teils der Entwicklung von eingebetteten Systemen möglich.
Die Fehlersuche im Entwicklungsprozess eines Systems kann deutlich reduziert werden und verkürzt dadurch zusätzlich noch den Produktentwicklungszyklus.

Ausblick

Zur Erweiterung und Verbesserung des Systems sind im Laufe der Arbeit diverse Ideen entstanden. Die signifikantesten werden im Folgenden zur Anregung für weitere Arbeiten kurz beschrieben.

Schnittstellenconstraints
Die Erweiterung der Schnittstellenspezifikation um zusätzliche Constraints, die die Anpassung der Kommunikation auf gegebene Register Files zulassen, wie es z.B. im ComiX-Ansatz umgesetzt wird, stellt eine sehr interessante Erweiterung dar. Bestehende Produkte könnten so erneut als SysML-Modell modelliert werden, was die weitere Produktpflege vereinfachen könnte.

Systemsynthese
Für die Industrie wäre es durchaus interessant eine vollständige Integration des SyCoGen in den Xilinx Synthese Flow vorzunehmen und als Konsequenz daraus die Generierung von

kompletten Xilinx ISE-/EDK-Projekten zu ermöglichen. Das bestehende System wüchse damit zur vollständigen Plattform für die Systemsynthese heran, die wiederum produktiv eingesetzt werden könnte.

Funktionelle Code-Generierung

Abschließend erwähnenswert ist die Tatsache, dass die Modellierung der Verhaltensbeschreibung für den Kommunikationskanal die interessanteste Erweiterung des bestehenden Systems darstellte. Eine Erweiterung der Typisierungsschicht um eine Applikationsschicht, die die genauen Inhalte der primitiven Datentypen definiert, wäre vonnöten und ermöglichte dann unter Verwendung von Aktivitäts- oder Zustandsmaschinendiagrammen die komplette Steuerung eines Peripheriebausteins zu modellieren bzw. zu generieren.

Literaturverzeichnis

[1] 19501:2005, ISO/IEC: *Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2*. 2005. – URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=32620

[2] FRIEDENTHAL, S. ; MOORE, A. ; STEINER, R.: *A Practical Guide to SysML: The Systems Modeling Language*. Elsevier Science & Technology, 2009 (The MK/OMG Press Series). – URL http://books.google.com/books?id=iNydHqO7HEEC. – ISBN 978-0-12378-607-4

[3] FRIEDENTHAL, Sanford ; GRIEGO, Regina ; SAMPSON, Mark: *INCOSE Model Based Systems Engineering (MBSE) Initiative*. INCOSE 2007 Symposium. June 2007. – URL http://www.incose.org/enchantment/docs/07docs/07jul_4mbseroadmap.pdf

[4] IBM: *CoreConnect Bus Architecture*. – URL https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4

[5] IHMOR, Stefan: *Modeling and Automated Synthesis of Reconfigurable Interfaces*, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, Dissertation, Januar 2006

[6] KECHER, Christoph: *UML 2 - Das umfassende Handbuch*. 3. Bonn : Galileo Press, 2009. – URL http://www.galileocomputing.de/1142?GPP=opoop. – ISBN 978-3-83621-419-3

[7] KOUDRI, Ali ; CHAMPEAU, Joël ; AULAGNIER, Denis ; SOULARD, Philippe: MoPCoM/MARTE Process Applied to a Cognitive Radio System Design and Analysis. In: PAIGE, Richard (Hrsg.) ; HARTMAN, Alan (Hrsg.) ; RENSINK, Arend (Hrsg.): *Model Driven Architecture - Foundations and Applications* Bd. 5562. Springer Berlin / Heidelberg, 2009, S. 277–288. – URL http://dx.doi.org/10.1007/978-3-642-02674-4_20

[8] NOBLE: *Datenblatt Rotary Enocers RE Series*. – URL http://www.pollin.de/shop/downloads/D240383D.PDF

[9] OMG: *OMG Systems Modeling Language (OMG SysML™)*. Object Management Group, June 2010. – URL http://www.omg.org/spec/SysML/1.2/PDF/

[10] OMG: *OMG Unified Modeling Language™(OMG UML), Infrastructure*. Object Management Group, May 2010. – URL http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/

[11] OPPENHEIMER, Frank: *OOCOSIM - an object-oriented co-design method for embedded hw/sw systems*. Uhlhornsweg 49-55, 26129 Oldenburg, Universität Oldenburg, Dissertation, 2005. – URL http://oops.uni-oldenburg.de/frontdoor.php?source_opus=172&la=en

[12] OPPENHEIMER, Frank: *The OSSS 2.2.0 Tutorial*. 2008. – URL http://system-synthesis.org/_media/osss2_tutorial.pdf

[13] VÖLTER, Markus: *Code Generation*. 2006. – URL http://www.voelter.de/data/presentations/mdsd-tutorial/07_CodeGeneration.pdf

[14] WEILKIENS, Tim: *Systems Engineering mit SysML/UML*. 2. Heidelberg : dpunkt.verlag GmbH, 2008. – URL http://www.system-modellierung.de. – ISBN 978-3-89864-577-5

[15] WIKIPEDIA: *Rotary encoder — Wikipedia, The Free Encyclopedia*. 2011. – URL http://en.wikipedia.org/w/index.php?title=Rotary_encoder&oldid=432830163. – [Online; accessed 22-June-2011]

[16] XILINX: *DS448 - PLB IPIF (v2.02a)*. 2005. – URL http://www.xilinx.com/support/documentation/ip_documentation/plb_ipif.pdf

[17] YUE, Minyi: A simple proof of the inequality FFD(L) $\leq$ 11/9 OPT + 1, $\forall$ L; for the FFD bin-packing algorithm. In: *Acta Mathematicae Applicatae Sinica (English Series)* 7 (1991), S. 321–331. – URL http://dx.doi.org/10.1007/BF02009683. – 10.1007/BF02009683. – ISSN 0168-9673

Anhang

Alle unten aufgeführten Anhänge sind in elektronischer Form auf der beigelegten CD-ROM zu finden.
Bei Interesse können diese auch direkt vom Autor mittels einer Anfrage per E-Mail an
fabian.zahn@googlemail.com
bezogen werden.

CD-ROM Struktur

- **Artisan Studio: SysML Modelle der Beispiele**
  (./Artisan Studio Models/Examples v0.zip)
  Die Beispieldiagramme und allgemeinen Modelle sind in diesem Archiv zusammengefasst.
- **Artisan Studio: SysML Modelle der Fallstudie**
  (./Artisan Studio Models/RotaryEncoder v0.zip)
  Alle Diagramme, die im Kapitel Realisierung verwendet wurden, sind in diesem Archiv enthalten.
- **Online-Quellen des Literaturverzeichnisses**
  (./Online Sources/*.pdf)
  Die im Literaturverzeichnis verwendeten Online-Quellen und Datenblätter sind in diesem Ordner als PDF-Dateien organisiert.
- **SyCoGen: Ausführbare Datei**
  (./SyCoGen/Jar/SyCoGen.jar)
  Der implementierte Codegeneratorprototyp als ausführbare JAR-Datei inklusive der Rotary Encoder XML-Beschreibung und den Template Files.
- **SyCoGen: Quellcode inkl. Eclipse-Projekt**
  (./SyCoGen/Source/Eclipse Project.zip)
  Der Quellcode des SyCoGen und die Templates, die zur Codegenerierung benötigt werden, sind innerhalb eines Eclipse-Projekts zu finden. Die Dokumentation des Quellcodes liegt ebenfalls in Form von Javadocs vor.

- **Bachelorthesis**
  (./Thesis/thesis.pdf)
  Dieses Dokument in elektronische Form als PDF-Datei.
- **Xilinx ISE Projekt: Fallstudie Rotary Encoder (EDK)**
  (./Xilinx ISE Projects/MyNexsys2Project.zip)
  Das Xilinx EDK Projekt des gesamten eingebetteten Systems der Fallstudie ist in diesem Archiv zusammengefasst.
- **Xilinx ISE Projekt: Rotary Encoder IP Core**
  (./Xilinx ISE Projects/MyRotaryEncoderInterface.zip)
  Der Kernel des Rotary Encoder Interfaces (IP Core) ist in diesem Archiv als Xilinx ISE Projekt vorhanden. Neben den Quellen ist auch eine Testbench enthalten.
- **Xilinx ISE Projekt: Register File Verifikation**
  (./Xilinx ISE Projects/RegisterFileVerification.zip)
  Das zur Verifikation des Register Files benötigte Testprojekt ist in diesem Archiv in Form eines Xilinx ISE Projekts zu finden.

Glossar

CASE **C**omputer-**A**ided **S**oftware **E**ngineering bezeichnet den durch Software-Werkzeuge unterstützten Entwurf von Software z.B. durch die graphische Modellierung oder Code-Generierung.

FPGA **F**ield **P**rogrammable **G**ate **A**rray ist ein Hardware-Baustein aus der Klasse der programmierbaren Logik.

HAL **H**ardware **A**bstraction **L**ayer bezeichnet eine Hardware-Abstraktionsschicht, die den Zugriff auf die darunterliegende Hardware durch Abstraktion erleichtert und so von der Bitmanipulationsebene trennt.

HDL **H**ardware **D**escription **L**anguage ist der Oberbegriff für Hardware-Beschreibungssprachen jeglicher Ausprägung. Die meist verbreiteten Hardware-Beschreibungssprachen sind Verilog und VHDL.

SoC **S**ystem **o**n (a) **C**hip bezeichnet die Integration sämtlicher Teilsysteme eines Systems auf einem einzigen Chip.

SoPC **S**ystem **o**n **P**rogrammable **C**hip bezeichnet das Gesamtsystem eines SoC, das auf einem programmierbaren Chip (z.B. ein FPGA) integriert wird.

SyCoGen **Sy**sML **Co**de **Gen**erator ist der Name des in dieser Arbeit implementierten Codegeneratorprototypen, der aus formalen XML-Beschreibungen Hardware-/Software-Schnittstellen synthetisiert.

SysML **Sys**tems **M**odeling **L**anguage ist die von der OMG definierte Spracherweiterung der UML, um die besonderen Anforderungen der Systemmodellierung hinzuzufügen.

UML **U**nified **M**odeling **L**anguage definiert eine von der OMG spezifizierte graphische Modellierungssprache für diverse Anwendungen.

VHDL **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage wird genutzt, um digitale Hardware formal zu beschreiben.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 26. August 2011	
Ort, Datum	Unterschrift