Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik

VHDL Codegenerierung für rekonfigurierbare Schnittstellen in eingebetteten Systemen

Studienarbeit

Studiengang Informatik

von

Gilles Bertrand Gnokam Defo
Andreasstr. 17
33098 Paderborn

betreut durch

Dipl.-Inform. Stefan Ihmor

vorgelegt bei

Prof. Dr. rer. nat. Franz Josef Rammig

im

Mai 2005

Dank und Erklärung

Dieses Dokument entstand im Rahmen einer Studienarbeit in der Arbeitsgruppe von Prof. Dr. rer. nat. Franz Josef Rammig (HNI) der Universität Paderborn.

Für das interessante Studienarbeitsthema möchte ich mich bei Prof. Franz J. Rammig bedanken. Besonderer Dank gilt Stefan Ihmor, der mich über den Zeitraum der Erstellung der Studienarbeit fachlich engagiert betreut hat. Nicht zuletzt möchte ich Anna Melinda Barát für die moralische Unterstützung danken.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Paderborn, 23.05.2005

Inhaltsverzeichnis

1. Einführung 1
1.1. Motivation . . . 1
1.2. Aufgabenstellung . . . 2
1.3. Gliederung . . . 2

2. Grundlagen 5
2.1. Pattern für die Codegenerierung . . . 5
2.1.1. Einsatzgebiete der Codegenerierung . . . 5
2.1.2. Pattern . . . 5
2.2. Interface Synthesis Design-Flow (IFS-Flow) . . . 10
2.2.1. System-Architektur . . . 10
2.2.2. Target Platform Description (TPD) . . . 10
2.2.3. Interface Description (IFD) . . . 11
2.2.4. Das IFD-Mapping . . . 11
2.2.5. IFS-Flow . . . 12
2.3. Der Interface Block (IFB) . . . 13
2.3.1. IFB-Makrostruktur . . . 14

3. IFB-Synthese 15
3.1. IFB-Rekonfiguration . . . 15
3.1.1. Rekonfiguration der Handler . . . 16
3.1.2. Steuerung der Rekonfiguration durch die Control Unit . . . 17
3.2. Erzeugung des Inputs für den Codegenerator . . . 19

4. Codegenerierung 21
4.1. Analyse des IFB-Templates . . . 21
4.1.1. Protocol Handler . . . 21
4.1.2. Sequence Handler . . . 22
4.1.3. Control Unit . . . 24
4.1.4. Ergebnis der Analyse des IFB-Templates . . . 24
4.2. Auswertung der Interface Descriptions (IFD) . . . 25
4.3. Aufbau des Codegenerators . . . 26
4.3.1. VHDL Gen . . . 26
4.3.2. VHDL Factory . . . 27

4.4. Erzeugung der VHDL-Pattern 27
4.5. Generierung einer Entity 27
4.6. Generierung einer Architecture 29
4.6.1. Generierung der Verhaltensbeschreibung 30
4.6.2. Generierung der Strukturbeschreibung 34
4.7. Synthese der Register-File 35
4.7.1. Frames und Packages 35
4.7.2. Register-Files 36

5. Implementierung des Codegenerators **37**
5.1. Package-Diagramm des Codegenerators 37
5.2. Klassendiagramme 38
5.2.1. vhdlGen 38
5.2.2. vhdlFactory 39
5.3. Verwendete Datenstrukturen 40
5.3.1. CodeList, CodeFragment, Tag 40
5.4. Automatisierung: CodegenerationWizard 40

6. Zusammenfassung und Ausblick **43**
6.1. Zusammenfassung 43
6.2. Ausblick 43

A. Anhang **45**
A.1. SynthesisWizard 45

Literaturverzeichnis **49**

Abbildungsverzeichnis

2.1. Template + Filtering . . . 6
2.2. Template + Metamodel . . . 6
2.3. Frame Processing . . . 7
2.4. API Based Generators . . . 8
2.5. Inline Code Generation . . . 8
2.6. Code Attributes . . . 9
2.7. Code Weaving . . . 9
2.8. Die System-Architektur . . . 10
2.9. IFS-Flow . . . 12
2.10. IFB-Makrostruktur . . . 13

3.1. Coarse- and Fine grained IFB Rekonfiguration . . . 15
3.2. SH mit einem Switch, einem Mode und zugehörigem Recon-Mode . . . 16
3.3. Gatterschaltung des Scoreboards in der Control Unit . . . 17
3.4. Rekonfigurationsablauf in der Control Unit . . . 18
3.5. Automatisierte IFB Generierung in zwei Syntheseschritten . . . 19

4.1. Der Protocol Handler und seine Komponenten . . . 22
4.2. Der Sequence Handler und seine Komponenten . . . 23
4.3. Verhalten- und Strukturbeschreibung einer Architektur . . . 24
4.4. VHDL-Codegenerierung durch Frame Processing . . . 26
4.5. Protocol-Matrix . . . 35

5.1. Codegenerator . . . 37
5.2. Package VhdlGen . . . 38
5.3. Package VhdlFactory . . . 39
5.4. CodegenerationWizard, Schritt 1 - Auswahl des IFBs . . . 41
5.5. CodegenerationWizard, Schritt 2 - Auswahl des VHDL Zielordners . . . 41
5.6. CodegenerationWizard, Schritt 3 - Generierung des VHDL-Codes . . . 42
5.7. Xilinx ISE 6.1 - Ausschnitt aus einer generierten VHDL-Beschreibung . . 42

A.1. Synthesis Wizard, Schritt 1 - Auswahl der zu verbindenden Schnittstellen 46
A.2. Synthesis Wizard, Schritt 2 - Auswahl der Zielplattform . . . 46
A.3. Synthesis Wizard, Schritt 3 - Definition der Datenabbildung und Festlegung der Rekonfigurierbarkeit des IFBs . . . 47

A.4. Synthesis Wizard, Schritt 4 - Erzeugung des IFBs 47
A.5. Synthesis Wizard, Schritt 5 - Einbinden des IFBs 48
A.6. Der generierte IFB im IFS-Editor 48

1. Einführung

1.1. Motivation

Der Begriff Codegenerierung bezeichnet die automatische Erzeugung von Quelltext in einer bestimmten Sprache. Codegeneratoren werden im Compilerbau und in der Sofwaretechnik eingesetzt. Ein Codegenerator im Compilerbau ist ein Teil eines Compilers, der während der Synthesephase eines Kompiliervorgangs Maschinencode, Assembler oder entsprechenden Code in einer anderen Sprache erzeugt. Auch in der Domäne der eingebetteten Systeme nimmt die Codegenerierung einen wichtigen Platz ein. Es gibt viele Gründe, die für eine automatische Codegenerierung sprechen:

- Falls eine Zielplattform nur über eine begrenzte Menge an Ressourcen verfügt, ist es sinnvoll, den Code in Abhängigkeit von den vorhandenen Komponenten zu generieren.
- Die Codierung erfolgt schneller und ist weniger fehleranfällig, wenn sie automatisiert erfolgt. Ebenso kann die Einhaltung bestimmter Code-Styles hierdurch leicht garantiert werden. Das fördert die Wiederverwendbarkeit und vereinfacht den Test des erzeugten Codes.
- In sicherheitskritischen Systemen, wie sie in eingebetteten Systemen häufig vorkommen, ist es besonders wichtig, den erzeugten Code validieren zu können. Ein Codegenerator kann daher den Aspekt "Design for testability" direkt als Teil der Code-Erzeugung berücksichtigen und so die Analyse des Codes erleichtern.

Die automatische Codegenerierung ist auch ein wichtiger Teil des *Interface Synthesis Design Flows* (IFS-Flow). Der IFS-Flow beschreibt eine Methodik der Modellierung und der automatisierten Synthese von rekonfigurierbaren Schnittstellen in eingebetteten Systemen. Das Ziel dabei ist es, einen *Interface Block* (IFB) zu erzeugen, der dazu genutzt werden kann, Anwendungen mit inkompatiblen Schnittstellen zu verbinden, ohne diese modifizieren zu müssen. Der IFS-Flow ist dazu in vier Schritte gegliedert. Zuerst erfolgt die Modellierung eines Kommunikationssystems. Hier werden die Anwendungen sowie deren Kommunikationsinfrastruktur beschrieben. Daran schließt sich die Synthese einer abstrakten IFB-Beschreibung an. Anschließend erfolgt die Generierung einer Implementierung des IFBs in einer konkreten Zielsprache. Der IFS-Flow endet mit der Integration des generierten IFBs in das existierende Design.

Die Modellierung im ersten Schritt des IFS-Flows ermöglicht die Beschreibung der Schnittstellen sowohl von Software- als auch von Hardware-Anwendungen. Der zu erzeugende IFB selbst kann auch in Software bzw. Hardware implementiert werden.

1.2. Aufgabenstellung

Die Aufgabe dieser Studienarbeit besteht darin, die Codegenerierung als dritten Schritt des IFS-Flows exemplarisch für die Sprache VHDL zu realisieren. Dabei soll ein VHDL-Codegenerator implementiert werden, der synthetisierbaren VHDL-Code für einen Interface Block generieren kann. Der Codegenerator soll dabei in der Lage sein, sowohl die rekonfigurierbare als auch die nicht-rekonfigurierbare Version des IFBs [Ihm05a] zu erzeugen.

Eine Aufgabe der Studienarbeit besteht darin, die Eingabe (-datenstruktur) des Codegenerators, die gleichzeitig die Ausgabe des ersten Syntheseschrittes ist, festzulegen. Dabei sind die Aspekte der IFB-Synthese, die von der zu erzeugenden Zielsprache abhängen, als zweiter Syntheseschritt direkt in die Codegenerierung zu integrieren.

Der IFS-Editor ist ein Software-Werkzeug, das eine grafische Oberfläche für den IFS-Flow implementiert. Der entwickelte Codegenerator ist als funktionale Komponente in den IFS-Editor zu integrieren. Auf diese Weise soll ein durchgängiger Entwurfsablauf von der Modellierung bis zur Erzeugung des endgültigen IFB-Codes ermöglicht werden. Zur Integration des IFBs in ein bestehendes Design kann der generierte Code mit Sprach- und Zielplattform-spezifischen Synthesewerkzeugen, wie z. B. “Xilinx-ISE” im Fall von VHDL, weiter verarbeitet werden.

1.3. Gliederung

Diese Studienarbeit ist in fünf Kapiteln gegliedert. Nach der Einführung folgt Kapitel zwei, in dem die Grundlagen vorgestellt werden, auf denen dieser Arbeit aufbaut. Dazu werden zuerst relevante Codegenerierungstechniken vorgestellt. Danach folgt eine Beschreibung des *Interface Synthesis Design Flows* sowie des generierten *Interface Blocks*.

In Kapitel drei werden die zur Rekonfiguration des IFBs eingesetzen Konzepte vorgestellt. Dabei wird an den wesentlichen IFB-Komponenten der Bezug der Rekonfiguration zur Codegenerierung hergestellt. Abschließend wird der erste Syntheseschritt des IFS-Flows diskutiert. Die Ausführungen beziehen sich hierbei im Wesentlichen auf die Erzeugung des IFB-Zwischenformats als Eingabe für den Codegenerator.

Der Aufbau und die Funktionalität des VHDL-Codegenerators werden in Kapitel vier beschrieben. Eine Analyse der IFB-Komponenten liefert das für diese Arbeit erforderliche Subset der Sprache VHDL. Anschließend wird erläutert, wie der aus dem ers-

ten Syntheseschritt erzeugte Input durch den Codegenerator ausgewertet wird. Die zur Übersetzung angewendete Technik des Frame-Processings wird anhand der wesentlichen VHDL-Elemente beispielhaft beschrieben.

Im fünften Kapitel wird die Java-Implementierung des Codegenerators vorgestellt.

Kapitel sechs fasst die erzielten Ergebnisse zusammen und liefert einen Ausblick auf weitere Arbeiten. Im Anhang wird die grafische Schnittstelle (GUI) des ersten Syntheseschrittes vorgestellt.

2. Grundlagen

2.1. Pattern für die Codegenerierung

Der Begriff Pattern steht für Muster oder Modell. Codegenerierung (bzw. Kodegenerierung) bezeichnet die automatische Erzeugung von Quelltext in einer bestimmten Programmiersprache [Wik]. Beispiele für Codegenerierung sind:

- Assembler, welche aus Assemblercode Maschinencode erzeugen
- Die Erzeugung von Quellcode aus einem Diagramm oder Modell, z.B. einem Programmablaufplan, einem Struktogramm oder einem UML-Modell
- Die Erzeugung der Implementierung einer abstrakten Beschreibung z.B. bei Application Server-Frameworks wie J2EE

2.1.1. Einsatzgebiete der Codegenerierung

Codegeneratoren werden im Compilerbau und in der Softwaretechnik angewandt. Im Compilerbau ist ein Codegenerator ein Teil eines Compilers, der während der Synthesephase eines Kompiliervorgangs Maschinencode oder entsprechenden Code in der zugehörigen Sprache erzeugt.

In der Softwaretechnik sind Codegeneratoren Computerprogramme, die Quellcode aus anderen Computerprogrammen erzeugen. Sie werden im Rahmen des Softwareerstellungsprozesses vor oder während des Kompiliervorgangs eingesetzt.

2.1.2. Pattern

Es existieren verschiedene Pattern für die Generierung von Quellcode. In diesem Abschnitt wird ein Überblick über sieben grundlegende Pattern gegeben.

Template + Filtering: Diese Technik wird in Abbildung 2.1 dargestellt und könnte beispielhaft dazu verwendet werden, um den Code für die Klassen eines UML Klassendiagramms zu generieren. UML Modelle werden typischerweise nach XMI

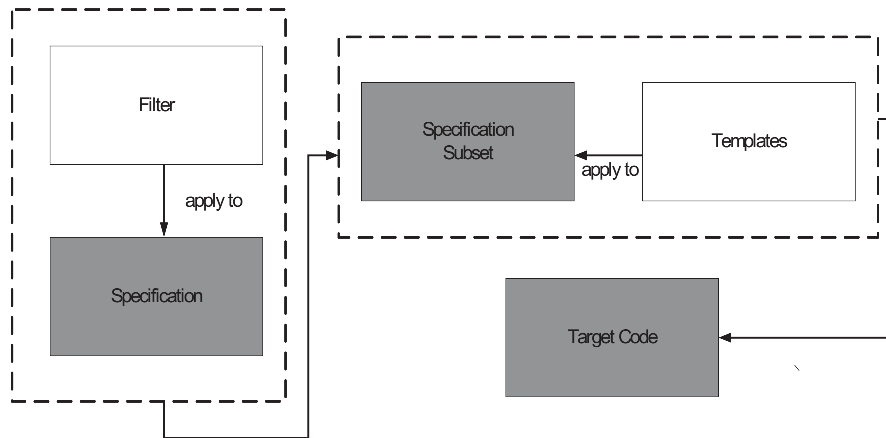

Abbildung 2.1.: Template + Filtering

Standards gespeichert (siehe [OMG03]). XMI Dateien sind typischerweise groß und komplex und beinhalten mehr Informationen als für die Generierung des Zielcodes notwendig sind. Die relevanten Subsets werden also aus dem Modell herausgefiltert. Dazu muss eine Syntax zum Filtern geschaffen werden, die diese relevanten Subsets selektiert. Anschließend werden Templates benutzt, um die Artefakte der Zielsprache beschreiben zu können. Diese Templates müssen ebenfalls in der Lage sein, auf die selektierten Teile des Modells zuzugreifen, um die Teile dann in den Zielcode einzufügen. Template + Filtering beschreibt eine recht einfache Technik der Codegenerierung. Diese ist besonders effizient, wenn man gute Mechanismen zum Filtern hat und die Spezifikationen wohldefiniert sind. Dennoch kann das Verfahren sehr kompliziert werden, wenn komplexe Filterregeln erforderlich sind. Eine Erweiterung dieses Verfahrens ist Template + Metamodel [Mar02].

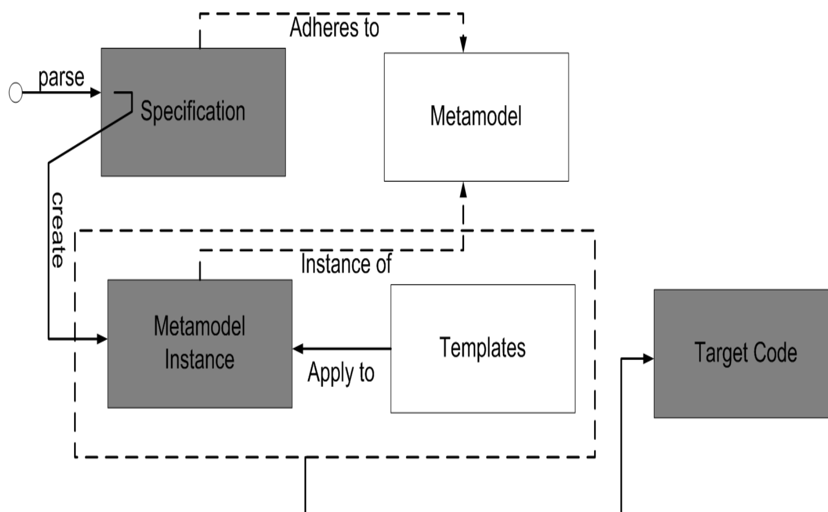

Abbildung 2.2.: Template + Metamodel

Template + Metamodel: Hier möchte man für die Produkte einer Software-Familie Code generieren. Template + Metamodel nutzt wohldefinierte Building Blocks, welche selbst wiederum eine Abbildung auf die Implementierungsplattform haben. Dies stellt sicher, dass die Templates für bestimmte Domänen spezifiziert werden können und so von Low-Level Details der Modellierung unabhängig sind. Somit stellt Template + Metamodel eine höhere Spezifikationsebene als Template + Filtering zur Verfügung.

Abbildung 2.2 zeigt, wie die Codegenerierung in zwei Schritten durchgeführt wird. Zuerst werden Metamodelle aus der Spezifikation instanziert und dann werden die Templates im Bezug auf diese Metamodelle geschrieben. Dadurch ist es einfacher die Spezifikation zu ändern, ohne ein Template anpassen zu müssen [Mar02].

Frame Processing: Bei dieser Technik werden parametrisierte Templates (auch Frame genannt) genutzt. Ein Frame kann als Funktion angesehen werden, welche durch ihre Ausführung Code generiert. Frame Processing wird im Abbildung 2.3 dargestellt. Um ein Software-System aus einer Sammlung von Frames zu erzeugen, können Frames durch *Slots* parametrisiert werden. Dabei bestehen Slots aus einer oder mehreren Frame-Instanzen oder Codeschnipseln wie zum Beispiel Typ, Name, etc. Die Codegenerierung wird durch einen Top-Level Frame kontrolliert. Dieser instanziert, parametrisiert und fügt die Instanzen aller Frames zusammen. Letztendlich evaluiert er die zusammengebaute Frame-Hierarchie, um den gewünschten Code (Target Code) zu generieren [Mar02].

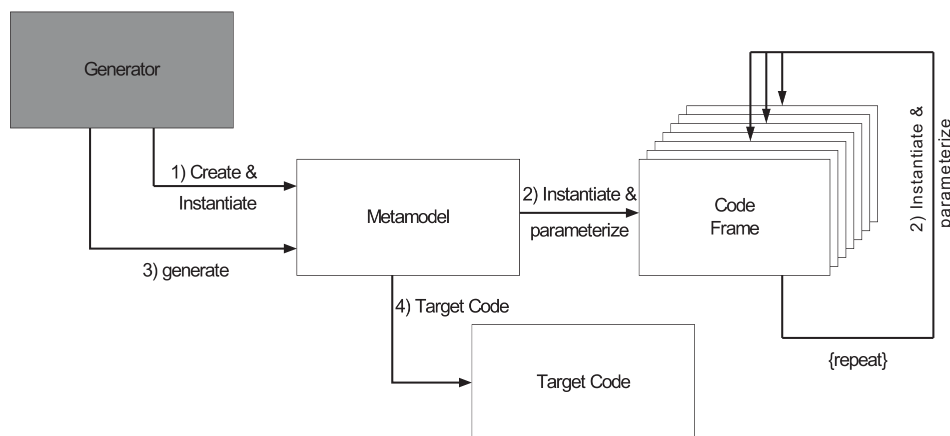

Abbildung 2.3.: Frame Processing

API-Based Generators: Bei dieser Technik wird eine API für die Codegenerierung zur Verfügung gestellt. Dadurch entsteht eine Abstraktion zu dem generierten Code. Unter Abstraktion ist zum Beispiel die konkrete Baum-Syntax des Zielcodes zu verstehen. Hier gibt es weder Templates noch Modelle. Stattdessen wird ein Client-Programm geschrieben, welches die API aufruft, um Code zu generieren bzw. zu modifizieren [Mar02]. Diese Technik wird im Abbildung 2.4 dargestellt.

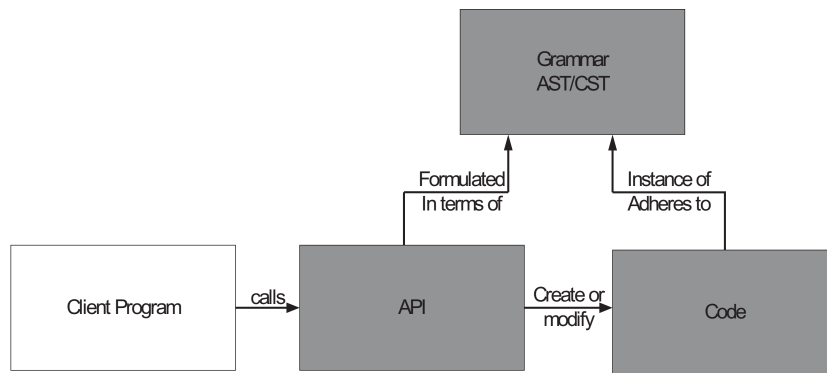

Abbildung 2.4.: API Based Generators

Inline Codegeneration: Inline Codegeneration ist eine Technik, um Code zu generieren, der bezüglich bestimmter Aspekte, wie zum Beispiel Datentypen oder Systemaufrufen, flexible ist. Wie Abbildung 2.5 dargestellt, müssen Code-Varianten im Source Code enthalten sein, damit während der Kompilierungsphase eine Konfiguration gewählt werden kann. Inline Codegeneration verwendet einen *Preprozessor*, der Bedingungen, Variablen, Typ-Ausdrücke, usw. während der Kompilierungsphase ersetzt, bevor diese ausgewertet werden. Der generierte Code wird so schrittweise zusammengesetzt und an einen spezifischen Kontext angepasst [Mar02].

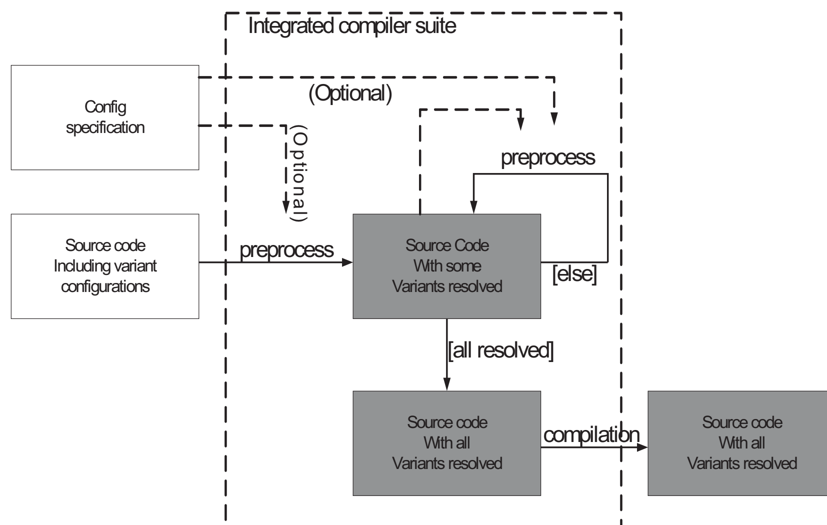

Abbildung 2.5.: Inline Code Generation

Code Attributes: Code Attributes ist eine Technik, die verwendet werden kann, um einen bereits existierenden Quellcode um zusätzliche Artefakte zu erweitern. Dabei greift das Verfahren auf Markierungen, die sogenannten Attribute, zurück, mit denen der Code versehen wurde. In den meisten Sprachen sind diese Attribute spezielle Kommentare. Der Codegenerator muss dann den Code und die Kommentare

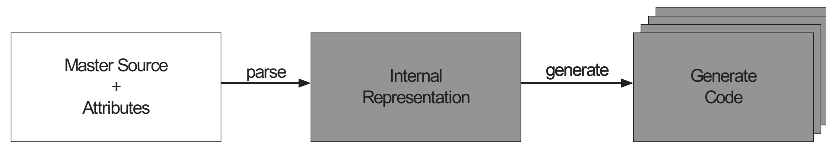

Abbildung 2.6.: Code Attributes

parsen, um die Erweiterung des Codes durchzuführen [Mar02]. Diese Technik wird im Abbildung 2.6 dargestellt.

Code Weaving: Der Begriff *Code Weaving* steht für *Code zusammensetzen* und wird in Abbildung 2.7 dargestellt. Bei dieser Technik werden verschiedene Teile des Programmtextes durch einen CodeWeaver in einer wohldefinierten Weise zusammengeführt. Die unterschiedlichen Teile werden als *Meta-Artefakte* bezeichnet und repräsentieren unterschiedliche Aspekte, welche in einer kontrollierten Art und Weise zu einer Anwendung verschmolzen werden. Dabei müssen Semantik und Zuständigkeit von jedem *Meta-Artefakt* klar definiert werden. Die "join-specification" definiert Abhängigkeiten, die beschreiben, wie die *(Meta-)Artefakte* zusammenpassen und sich untereinander beeinflussen [Mar02].

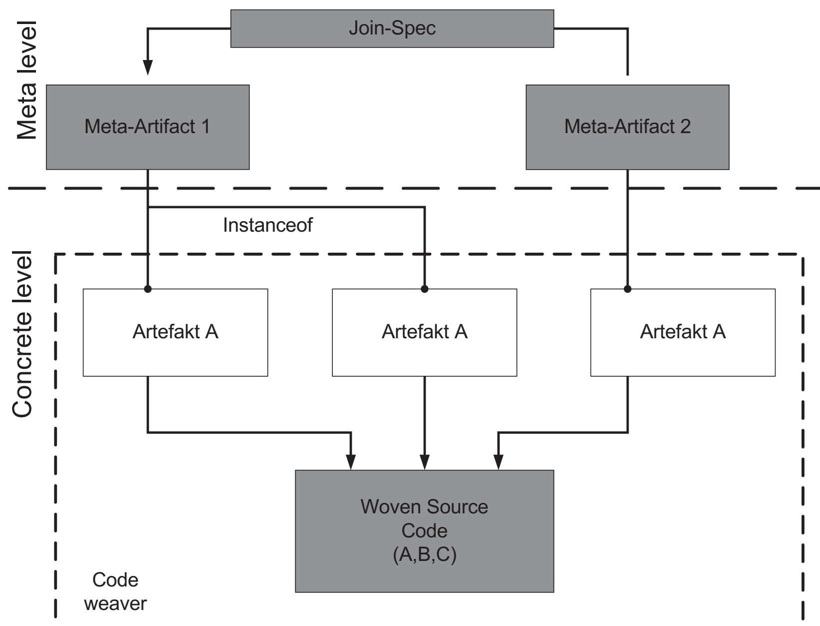

Abbildung 2.7.: Code Weaving

2.2. Interface Synthesis Design-Flow (IFS-Flow)

Der Interface Synthesis Design Flow (IFS-Flow) ist eine Methodik, die für die Modellierung und automatisierte Synthese von rekonfigurierbaren Schnittstellen in eingebetteten Systemen definiert wurde. Die flexible Wiederverwendbarkeit und Anpassung von inkompatibeln IPs (Intellectual Property) ist ein integraler Teil des Entwurfsprozesses. Um den IFS-Flow besser zu verstehen, sollen zunächst die Begriffe System-Architektur, Target Platform Description (TPD) und Interface Description (IFD) eingeführt werden.

2.2.1. System-Architektur

Die *IFS-System-Architektur* wurde für die Entwicklung komplexer Kommunikationsprototypen erdacht. Sie hat, wie im Abbildung 2.8 dargestellt, eine hierarchische Struktur, die aus folgenden *System-Komponenten* besteht: *System*, *Board*, *Chip*, *Task* und *Medium*. Die Verbindungen zwischen den Komponenten repräsentieren elektrische Verbindungen. Die System-Komponenten sind unterteilt in *Architektur-Komponenten* (System, Board, Chip) und *Kommunikations-Komponenten* (Task, Medium). Im Unterschied zu Medien, sind Tasks darauf beschränkt ausschließlich auf einem Chip positioniert zu sein. Beide Arten von Komponenten verfügen über Schnittstellenbeschreibungen (Interface Description). Zusätzlich beinhalten die Kommunikationskomponenten Verhaltensbeschreibungen in Form von Protokoll-Zustands-Automaten.

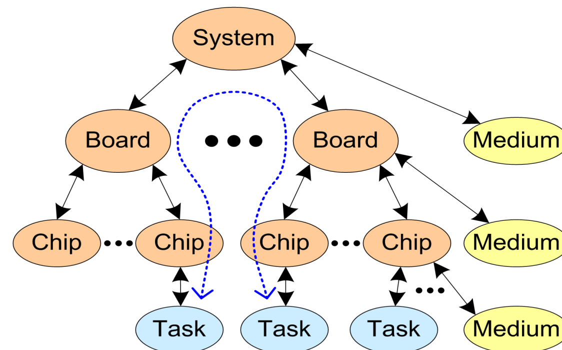

Abbildung 2.8.: Die System-Architektur

2.2.2. Target Platform Description (TPD)

Die Target Platform Description (TPD) beschreibt die Eigenschaften der Zielplattform, auf der der gernerierte Target-Code des IFBs ausgeführt wird. Der zu erzeugende VHDL-Code muss deshalb unter Berücksichtigung dieser Eigenschaften generiert werden. In der TPD werden alle verfügbaren Clocks und Ressourcen beschrieben, die für die Synthese relevant sind. Ressourcen können z. B. die Complex Logic Blocks (CLBs) eines Field Programmable Gate Arrays (FPGAs) oder der Speicher eines Prozessors sein.

2.2.3. Interface Description (IFD)

Eine Interface Description (IFD) ist ein Container, der durch ein Tripel (I-P-M) definiert wird, die drei Elemente die er beinhalten kann sind:

- Ein *Interface* (I) beschreibt eine Menge von physikalischen Pins. Dazu beinhaltet ein Interface die Attribute eines Pins, wie Richtung, Typ, Bitbreite, etc. Weiterhin beschreibt ein Interface die elektrischen Eigenschaften eines Pins. Diese dienen bisher ausschließlich dem Test, ob zwei Pins miteinander verbunden werden dürfen. Die elektrische Signalumsetzung ist nicht Gegenstand des IFS-Flows.
- Ein *Protocol* (P) definiert eine Menge von virtuellen Pins sowie deren Verhalten. Das Verhalten wird durch Protokollzustandsautomaten beschrieben, dessen Ausgaben in den einzelnen Zuständen die jeweiligen Werte der Protokollpins festlegen.
- Eine *ProtocolMap* (M) wird verwendet, um die virtuellen Pins des Protocols auf die physikalischen Pins des Interfaces abzubilden.

2.2.4. Das IFD-Mapping

Ein IFD-Mapping besteht aus Mapping Equations und spezifiziert die Umsetzung der Informationen zwischen den kommunizierenden Tasks innerhalb eines IFBs. Die Mapping Equations $MapEqn$ beschreiben dazu die Transformation von eingehenden auf ausgehende Datenpakete mit Hilfe einer Mapping Funktion f_{Map} in der folgenden Form:

$$MapEqn : data_{Out} <= f_{Map}(data_{In,1}, ..., data_{In,k});$$

Ein $data_{Out}$-Paket kann dabei von 0 bis k $data_{In}$-Paketen abhängig sein. Um diese Abhängigkeiten zu definieren, bietet die *IFD-Mapping Sprache* vier Grundoperationen. Diese Operationen können auch miteinander kombiniert werden.

1. Zuweisung konstanter Werte
2. Zuweisung umsortierter eingehender Bits
3. Anwenden von boolschen Funktionen auf eingehende Bits
4. Verwenden eines endlichen Zustandsautomaten (FSM) zum Erzeugen von Bits

Um die IFD-Mappings mehrerer Tasks in einem IFB zu bearbeiten, kann ein so genannter *Multi-Task IFB* konstruiert werden. Das Zuordnen (Binding) eines IFD-Mappings zu einem IFB bedeutet das Implementieren des IFD-Mappings in diesem IFB [Ihm05b].

2.2.5. IFS-Flow

Wie Abbildung 2.9 zeigt, besteht der IFS-Flow aus vier Schritten:

1. Modellierung der System-Architektur
2. Syntheseschritt 1: Generierung des IFBs in einem *Zwischenformat*
3. Syntheseschritt 2: Generierung des IFBs in der *Zielsprache* (hier VHDL)
4. Der generierte IFB wird in das bereits implementierte Design integriert

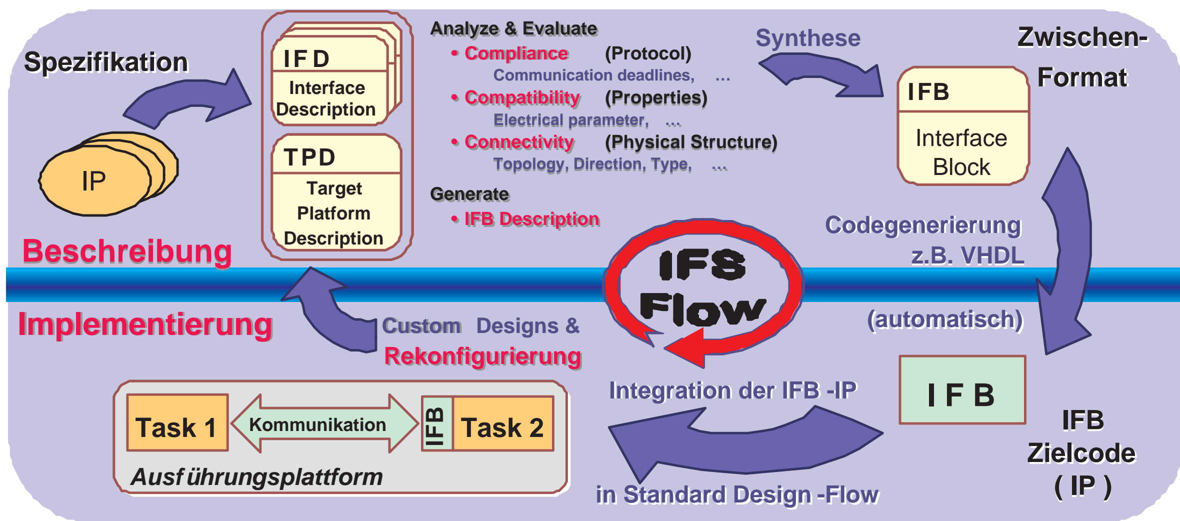

Abbildung 2.9.: IFS-Flow

Im ersten Schritt des IFS-Flows geht man von einer abstrakten Spezifikation einer System-Architektur aus, welche durch eine Menge von IPs (Intellectual Property) modelliert ist. Danach wählt der Designer die Schnittstellen Beschreibungen (IFD) der Tasks oder Medien, die verbunden werden sollen sowie die Zielplattform (TPD). Als nächstes folgt eine Phase, in der jede ausgewählte IFD bezüglich folgender Eigenschaften geprüft wird:

- Compliance: Einhaltung bestimmter Protokolleigenschaften, wie Deadlines, etc.
- Kompatibilität: Die Prüfung der elektrischer Parameter des Interfaces.
- Konnektivität: Test der physikalische Struktur, z. B. Typ und Richtung der Pins.

Sind alle Eigenschaften erfüllt, so kann eine einfache Verdrahtung zwischen den gewählten Komponenten erstellt werden. Andernfalls wird einen Interface Block als Adaptermodul synthetisiert. Eine Voraussetzung dafür ist jedoch, dass die Signale elektrisch kompatibel sind, da eine elektrische Umsetzung spezielle Hardware erfordern würde.

Das Synthetisieren eines IFBs erfolgt in zwei Schritten. Der erste Schritt dieser Synthese erzeugt eine abstrakte Instanz des IFBs in ein *Zwischenformat* auf der Beschreibungsebene. Dieser abstrakte IFB kann in Form einer XML IP sowohl importiert als auch exportiert werden. Im zweiten Syntheseschritt wird der endgültige IFB implementiert, d. h. er wird in einer Zielsprache durch eine Codegenerierung als Hardware oder Software Lösung erzeugt.

Im letzten Schritt des IFS-Flows wird die erzeugte IFB-Instanz in eine zuvor existierenden Implementierung der System-Architektur eingebettet [Ihm05b].

2.3. Der Interface Block (IFB)

Nachdem im vorigen Kapitel die Erzeugung eines IFBs beschrieben wurde, soll im Folgenden die Funktionalität in Verbindung mit dem Aufbau eines IFBs näher erläutert werden. Dies ist für die Codegenerierung von besonderer Bedeutung, da der Codegenerator eben diese Struktur in der Zielsprache konstruieren muss. Eine ausführliche Analyse einzelner Komponenten des IFBs erfolgt in Kapitel 4.

Der Interface Block ist ein Adapter Modul, das entwickelt wurde, um die Prokollübersetzung zwischen Anwendungen mit inkompatiblen Protokollen zu ermöglichen. Die allgemeine Realisierung eines IFBs wird durch die *IFB-Makrostruktur* beschrieben. Die spezielle Realisierung in Hardware ist durch das so genannte *IFB-Template* modelliert.

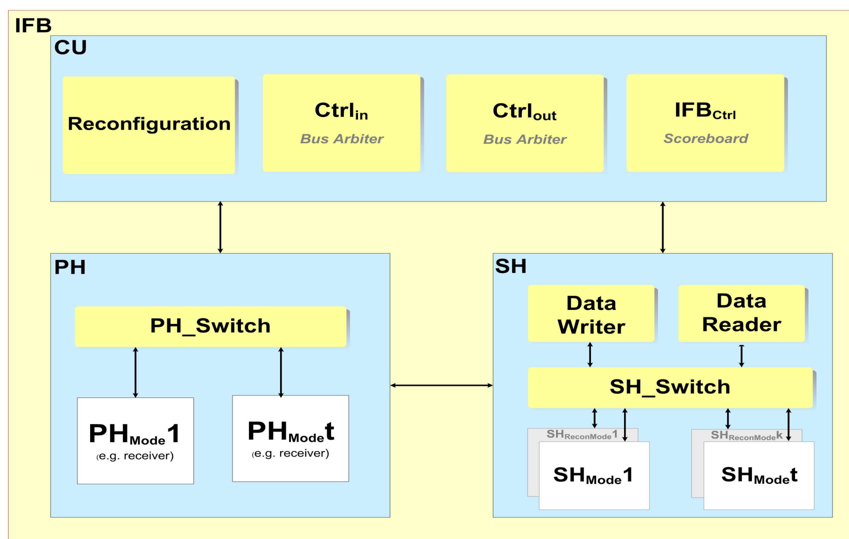

Abbildung 2.10.: IFB-Makrostruktur

2.3.1. IFB-Makrostruktur

Die Grundidee der IFB-Makrostruktur ist die Unterteilung des IFBs in drei Komponenten: den *Protocol Handler* (PH), den *Sequence Handler* (SH) und die *Control Unit* (CU). Abbildung 2.10 visualisiert die IFB-Makrostruktur. Der Protocol Handler und der Sequence Handler beinhalten jeweils eine Menge an Handlern. Jeder Handler besteht aus einer Menge von rekonfigurierbaren Stubs, die im Rahmen des IFS-Projektes als Modes bezeichnet werden. Für jede verbundene Schnittstelle einer Task existiert ein Protocol Handler Mode (PH_M) innerhalb des Protocol Handlers.

Das Verhalten innerhalb eines Modes wird durch einen endlichen Protokollzustandsautomaten (Finite State Machine (FSM)) implementiert. Ein PH_M implementiert das komplementäre Protokoll der korrespondierenden Schnittstelle. Dazu wird die komplementäre FSM automatisch aus der Schnittstellenbeschreibung (IFD) der verbundenen Task bzw. des Mediums abgeleitet [Ihm03]. Die Protokoll FSMs werden um zusätzliche Zustände erweitert, welche die Interaktion mit der Control Unit und dem Sequence Handler ermöglichen. Der Sequence Handler implementiert das IFD-Mapping. Dabei existiert für jede Mapping Equation ein Sequence Handler Mode (SH_M) im Sequence Handler zur Transformation der Nutzdaten.

Der Ablauf einer minimalen Kommunikation über den IFB sieht wie folgt aus: Ein PH_M extrahiert die Nutzdaten aus seinem Protokoll, die im IFD-Mapping als Input-Daten verwendet werden. Alle anderen Daten werden ignoriert. Die gelesenen Nutzdaten werden in den Speicher für eingehende Daten im SH gepuffert, durch die SH_M modifiziert und in den Speicher für ausgehende Daten im SH eingelagert. Anschließend werden die modifizierten Daten durch einen PH_M in ein ausgehendes Protokoll integriert. Die Koordination der FSMs innerhalb des IFBs wird durch die Control Unit (CU) gewährleistet.

3. IFB-Synthese

Dieses Kapitel soll einen Überblick über den ersten Syntheseschritt des IFS-Flows geben und die Bedeutung der Rekonfiguration verdeutlichen. Der erste Syntheseschritt ist für diese Arbeit bedeutend, da er den Input für die Codegenerierung in Form einer abstrakten IFB-Beschreibung erzeugt. Als ein Aspekt der Synthese wird das Konzept der IFB-Rekonfiguration betrachtet, das sich bis in die Codegenerierung auswirkt.

3.1. IFB-Rekonfiguration

Die IFB-Makrostruktur wurde hinsichtlich des Hardware-Entwurfes konzipiert. Um die Rekonfiguration unterstützen zu können, gibt es neben der "kompakten" Version des IFBs eine "partiell-rekonfigurierbare" Version für den rekonfigurierbaren Fall. Die Rekonfiguration wird deshalb als partiell bezeichnet weil, die für diesen Fall ausgewählte Ausführungsplattform, ein FPGA (Field-Programmable Gate-Array), partiell rekonfiguriert werden kann. Der Aufbau und die Funktionalität des IFBs im nicht-rekonfigurierbaren Fall wurde bereits im Kapitel 2 beschrieben. Im Folgenden wird die erweiterte Funktionalität im rekonfigurierbaren Fall erläutert. Dazu wird zunächst betrachtet, auf welche Weise der IFB rekonfiguriert werden kann.

Es gibt zwei verschiedene Varianten für die Rekonfiguration des IFBs: *Coarse grained* und *Fine grained* (siehe Abbildung **??**). Die *Coarse grained* Variante (auf der linken Seite der Abbildung zu sehen) erlaubt es, eine ganze Task einschließlich aller dazugehörigen *Interfaces* mit Hilfe des IFS-Flows auszutauschen. Die *Fine grained* Variante

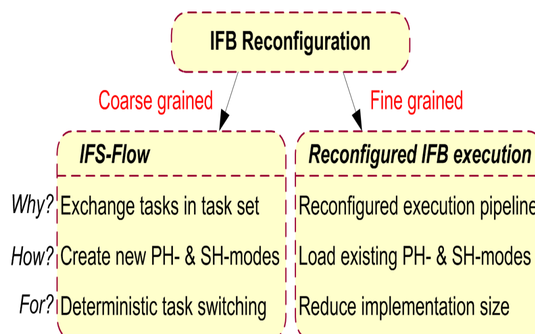

Abbildung 3.1.: Coarse- and Fine grained IFB Rekonfiguration

betrachtet die rekonfigurierte Ausführung des IFBs auf Ebene der Handler. Dazu wird vor Beginn der Laufzeit ein statisches Scheduling erstellt [Ihm05a]. Alle existierenden *Modes* werden zur Laufzeit erst dann geladen, wenn sie ausgeführt werden sollen. Dadurch wird die Größe der IFB-Implementierung reduziert. Um die Rekonfiguration des IFBs zu ermöglichen, beinhalten die Handler und die Control Unit zusätzliche Architekturkomponenten.

3.1.1. Rekonfiguration der Handler

In der rekonfigurierbaren Version des IFBs, wird zu jedem Sequence Handler Mode (SH_M) ein zusätzlicher Recon-Mode ($SHRecon_{Mode}$) erzeugt. Ein Recon-Mode wird dazu benötigt, ein deterministische Verhalten des Systems während der Rekonfiguration des zugehörigen SH_M aufrecht zu erhalten. Der $SHRecon_{Mode}$ wird automatisch aktiviert, sobald der zugehörige SH_M deaktiviert wird. Um Modes zur Laufzeit austauschen zu können, wird der *Switch* im Sequence Handler und im Protocol Handler um Tristate-Buffer für jedes vorhandene Signal erweitert. Mit Hilfe dieser Tristate-Buffer können die *Modes* gezielt durch die Control Unit deaktiviert bzw. aktiviert werden.

Abbildung 3.2 zeigt einen Ausschnitt des Sequence Handlers aus dem IFB-Template. Dargestellt ist der Switch, ein Mode sowie der zugehörige Recon-Mode. Der Switch beinhaltet die Tristate-Buffer zum physikalischen Abtrennen der Modes. In den Modes sind die Automaten zur Transformation der Daten während des Normalbetriebs bzw. der Rekonfiguration beschrieben.

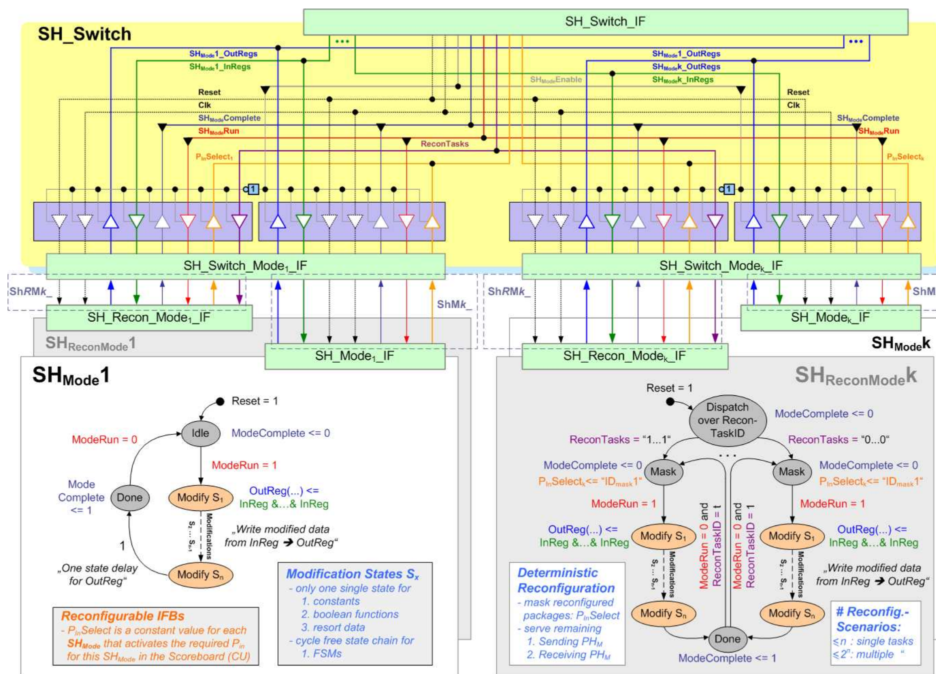

Abbildung 3.2.: SH mit einem Switch, einem Mode und zugehörigem Recon-Mode

3.1.2. Steuerung der Rekonfiguration durch die Control Unit

Wie bereits erwähnt wurde, ist die Control Unit für die Koordination der internen Kommunikation zuständig. Sie besteht grundsätzlich aus den beiden Schedulern $Ctrl_{In}$ und $Ctrl_{Out}$ und einem *Scoreboard*. Die *Scheduler* arbitrieren die IFB internen Busse für den Datenaustausch zwischen SH und PH. Dafür beziehen sie sich auf Informationen, die aus dem *Scoreboard* resultieren. Das *Scoreboard* verwaltet die Belegung des Speichers mit Datenpaketen im SH und aktiviert die SH Modes, wenn Daten zur Berechnung bereitstehen.

Die Schaltung des *Scoreboards* wird im rekonfigurierbaren Fall um zusätzliche Komponenten erweitert. Diese Erweiterungen sind in der Gatterschaltung in Abbildung 3.3 durch die grünen Kreise hervorgehoben. Im rekonfigurierbaren Fall haben alle Modes generell die Möglichkeit auf alle Datenpakete zuzugreifen. Dabei bleibt es jedem Mode selbst überlassen, sich die Datenpakete auszuwählen, die für seine Ausführung erforderlich sind. Dies geschieht durch das Signal P_{In}select. Da die Anzahl von Datenpakete variabel ist, ermöglicht P_{In}select zur Laufzeit die nicht verwendeten Datenpakete zu maskieren. Die Erweiterung des Scoreboards, die in Abbildung 3.3 auf der linken Seite dargestellt ist, verhindert das Lesen oder Schreiben von weiteren Datenpaketen der Modes, die von der Rekonfiguration betroffen sind.

Neben den drei bereits angesprochenen Komponenten der CU wird noch eine *Reconfiguration* Einheit in die CU integriert. Diese regelt den Ablauf der Rekonfiguration und besteht aus einem Zustandsautomaten.

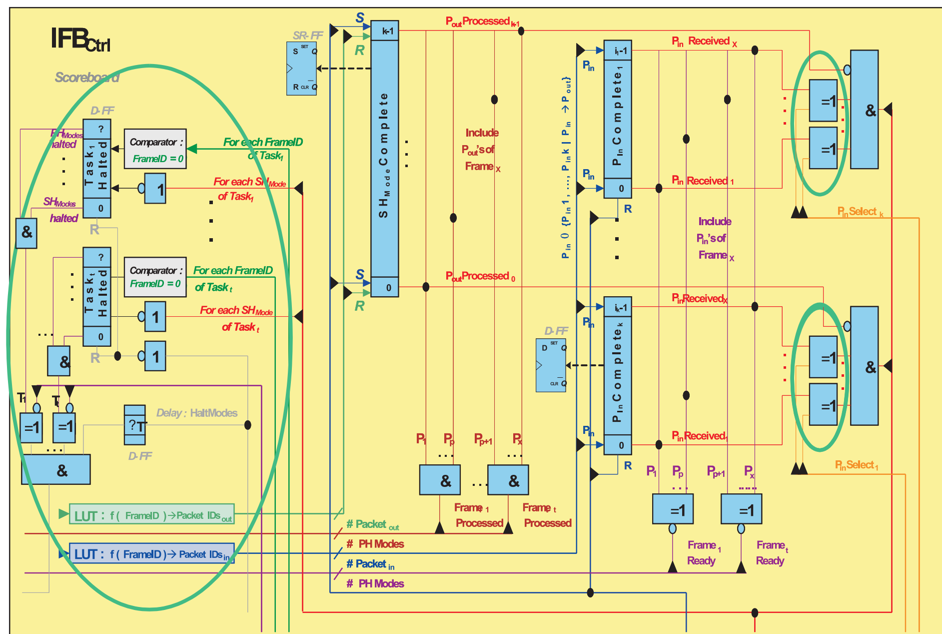

Abbildung 3.3.: Gatterschaltung des Scoreboards in der Control Unit

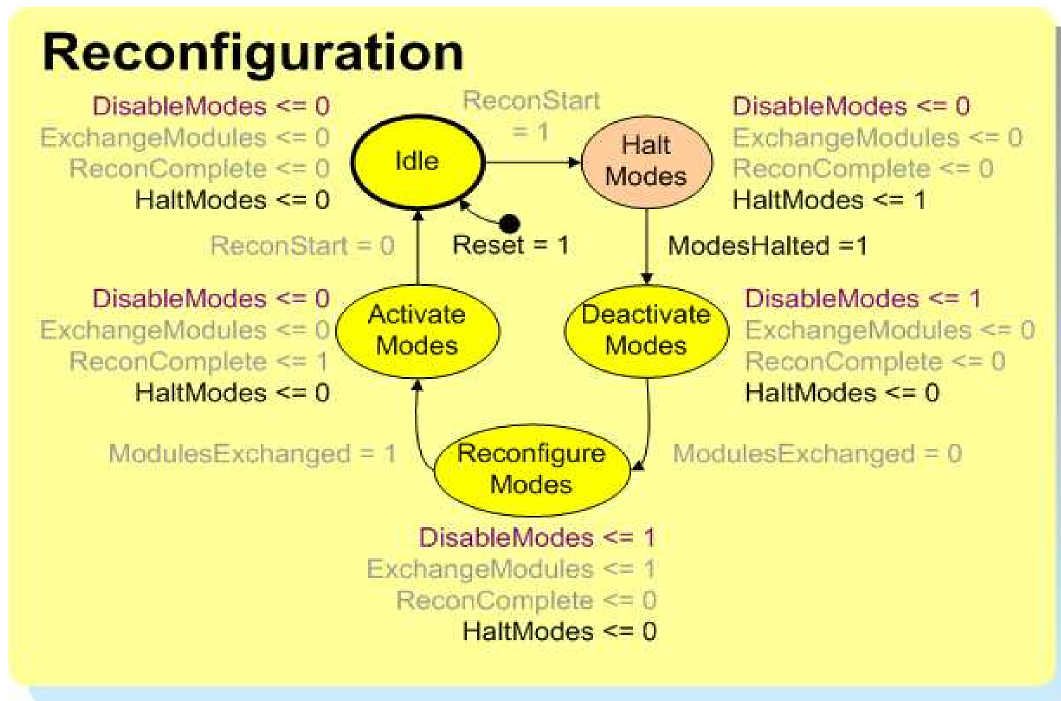

Abbildung 3.4.: Rekonfigurationsablauf in der Control Unit

Wie im Abbildung 3.4 dargestellt wird, erfolgt die Rekonfiguration des IFBs in vier Schritten:

1. Zuerst werden alle von der Rekonfiguration betroffenen Modes angehalten, so dass keiner dieser Modes mehr Datenpakete liest oder schreibt. Die dazu notwendige Erweiterung des Scoreboards wurde zuvor erläutert.

2. Die betroffenen Modes werden durch die Tristate-Buffer in den Switches deaktiviert.

3. Die eigentliche Durchführung der Rekonfiguration findet durch eine Komponente außerhalb des IFBs statt.

4. Die rekonfigurierten PH_{Modes} und SH_{Modes} werden zunächst physikalisch durch die Tristate-Buffer verbunden und anschließend wieder aktiviert.

Wie in den Abbildungen 3.3 und 3.4 ersichtlich wird, beinhaltet das IFB-Template Aspekte, die sich in VHDL sinnvoll als Strukturbeschreibung darstellen lassen und andere, die sich eleganter als Verhaltensbeschreibung modellieren lassen. Dies wird bei der Codegenerierung, wie sie in Kapitel vier vorgestellt wird, entsprechend berücksichtigt.

Um die Codegenerierung besser verstehen zu können, soll zunächst der erste Syntheseschritt des IFS-Flows näher erläutert werden, da die Ergebnisse dieser Synthese die Eingabe für die Codegenerierung darstellen.

3.2. Erzeugung des Inputs für den Codegenerator

Abbildung 3.5 präsentiert den Ablauf zur Erzeugung eines IFB bestehend aus zwei Syntheseschritten. Der erste Schritt erzeugt eine abstrakte Zwischenrepräsentation des IFBs (IFB-Description) die unabhängig von der eigentlichen Implementierung ist. Ein Teil der Arbeit bestand darin, den Inhalt und das Format der IFB-Description mitzugestalten. Der zweite Syntheseschritt setzt die IFB-Description in eine endgültige Implementierung in einer bestimmten Sprache um. Daher wird der zweite Syntheseschritt als Codegenerierung bezeichnet.

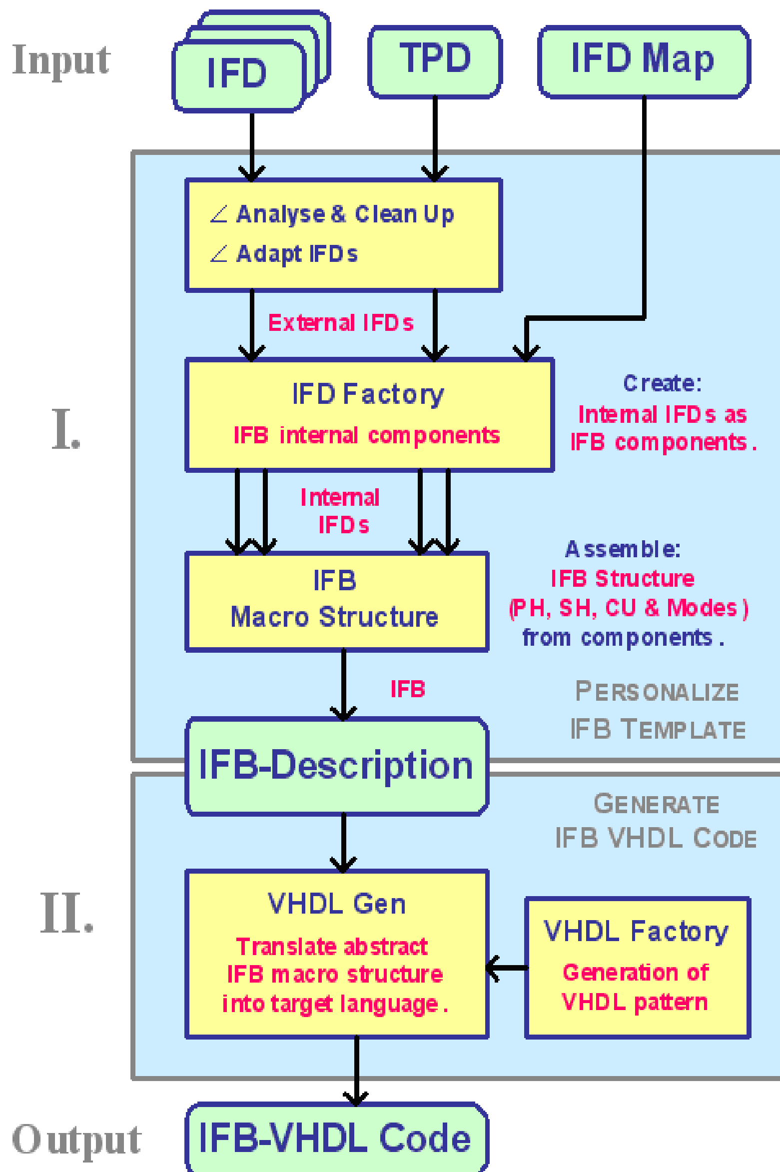

Abbildung 3.5.: Automatisierte IFB Generierung in zwei Syntheseschritten

Im ersten Syntheseschritt werden zuerst die in den Syntheseprozess eingehenden Daten analysiert und aufgearbeitet. In diesem Schritt werden überflüssige sowie fehlerhafte Einträge aus den eingehenden Beschreibungen gelöscht. Das eingesetzte Verfahren deckt sowohl den Aspekt der Strukturanalyse ("Gültigkeit der Verdrahtung"), als auch den der Verhaltensanalyse (z. B. dead-state elimination) ab. Die Bezeichner innerhalb der Beschreibungen werden automatisch an eine IFB interne Konvention angepasst. Um die komplementären Protokolle des Interface Blocks zu erzeugen, werden zuerst die Signalrichtungen der gegebenen Protokollbeschreibungen invertiert. Anschließend werden komplexe Ausdrücke innerhalb der Protokollbeschreibung aufgelöst ("flatten"), um eine hierarchielose und homogene Beschreibungsstruktur für den Syntheseprozess zu erhalten.

Danach werden mit Hilfe der *IFD-Factory* die Komponenten der IFB-Makrostruktur generiert. Da im zweiten Syntheseschritt, der Codegenerierung, ausschließlich IFDs verarbeitet werden können, kodiert die *IFD-Factory* alle Informationen in Form von IFDs. Dabei werden die eingehenden IFDs basierend auf dem *IFD-Mapping* um Zustände für die IFB interne Kommunikation erweitert. Aus diesen IFDs werden dann in der Codegenrierung die PH_{Modes} erzeugt. Weiterhin werden in der Factory die IFDs für den Sequence Handler und die Control Unit synthetisiert. Dazu werden das IFD-Mapping und die IFDs der angeschlossenen Tasks ausgewertet. Für die Zwischenspeicherung der Daten im SH werden die im IFD-Mapping beschriebenen Pakete von der Factory in eine abstrakte Datenstruktur umgeformt, die ein Register-File beschreibt.

Zur Erzeugung der IFB-Makrostruktur werden anschließend die Komponenten des IFBs wie in einem Baukastensystem aus den generierten IFDs zusammengesetzt. Dazu zählen der IFB selbst, der Protocol Handler mit Switch und PH_{Modes}, der Sequence Handler mit *DataReader*, *DataWriter* und den SH_{Modes} sowie die Control Unit und ihrer Komponenten. Je nachdem, ob der IFB rekonfigurierbar oder hierarchisch aufgebaut sein soll, variiert hierbei die Anordnung der Modes. Für einen rekonfigurierbaren IFB sind alle Modes außerhalb des IFBs zu platzieren. Dementsprechend muss die Verbindung zu den Switches anders gestaltet werden. Die Auflage, dass alle rekonfigurierbaren Komponenten auf Ebene des Top-Designs angeordnet sein müssen, leitet sich aus dem "partial design flow" für FPGAs der Firma Xilinx Inc. ab. Für die automatisierte Synthese wurde eine GUI (Graphical User Interface) namens *Synthesis Wizard* implementiert, die den Benutzer durch die einzelnen Stufen dieses Syntheseschrittes leitet. Der *Synthesis Wizard* wird im Anhang dieser Arbeit kurz vorgestellt.

Da die IFB-Description in XML nicht in ihrer endgültigen Form, sondern nur als Menge der für die Synthese notwendigen Eingabedaten abgespeichert wird, erfolgt die IFB-Synthese auch beim Laden eines IFBs. Dieses Vorgehen erhöht die Ladezeit eines IFBs, reduziert allerdings drastisch die Größe des Austauschformates der IFB-Description. Für den Anwender ist die Synthese während des Ladens nicht sichtbar.

Die Codegenerierung arbeitet direkt auf der Datenstruktur der IFD-Description, die im ersten Syntheseschritt im IFS-Editor aufgebaut wird. Um einzelne Ausschnitte der IFDs zu visualisieren, werden diese jedoch in ihrer XML Repräsentation dargestellt.

4. Codegenerierung

Im Kapitel 2 wurden sieben Techniken für Codegenerierung vorgestellt. Für die Implementierung des Codegenerators wird das Konzept des *Frame Processings* angewendet. Dieses Verfahren ist für die Generierung von VHDL-Code besonders geeignet, da VHDL eine Komponenten-orientierte Sprache ist. Das bedeutet, dass einzelne Teile von VHDL einfach auf Frames abgebildet werden können. In diesem Kapitel wird zunächst der Aufbau des Codegenerators näher erläutert. Dabei wird gezeigt, wie das Konzept des *Frame Processings* angewendet wird, um die IFB-Description auszuwerten. Die Erzeugung der IFB-Description wurde bereits zuvor im Kapitel 3 beschrieben.

4.1. Analyse des IFB-Templates

Ein IFB kann prinzipiell sowohl als Software- als auch als Hardware-Lösung realisiert werden. Der im Rahmen dieser Arbeit entwickelte VHDL-Codegenerator erlaubt es, den synthesefähigen VHDL-Code sowohl für einen rekonfigurierbaren als auch für einen nicht rekonfigurierbaren IFB zu generieren. Die Realisierung eines IFBs in Hardware wird durch ein 'Schematic' namens *IFB-Template* beschrieben. Im Folgenden soll eine Betrachtung ausgewählter Komponenten des IFB-Templates repräsentativ zeigen, welche VHDL-Pattern zu generieren sind. Die detaillierte Beschreibung des vollständigen Templates auf Schaltungsebene ist zum Verständnis der Codegenerators hierbei nicht notwendig und würde den Rahmen dieser Studienarbeit überschreiten.

4.1.1. Protocol Handler

Wie in Abbildung 4.1 dargestellt, besteht ein Protocol Handler aus einem Switch und einer Menge von Protocol Handler Modes. Die Modes implementieren Protokollzustandsautomaten und werden im IFB-Template durch eine abstrakte Darstellung von FSMs modelliert. Der Codegenerator erzeugt die FSMs in Form von VHDL-Verhaltensbeschreibungen. Eine beispielhafte Ausprägung des Switches ist im IFB-Template auf Gatterebene modelliert. Die endgültige Form des Switches lässt sich direkt aus der IFB-Description ableiten. Die daraus resultierende Schaltung könnte in VHDL sowohl als Struktur- als auch Verhaltensbeschreibung erzeugt werden. Aufgrund der kompakteren Darstellung wurde sich hierbei für die Verhaltensbeschreibung entschieden. Dabei sind im Wesentlichen einfache Logik-Gatter und Tristate-Buffer zu erzeugen.

Abbildung 4.1.: Der Protocol Handler und seine Komponenten

4.1.2. Sequence Handler

Wie der Protocol Handler besteht auch der Sequence Handler aus einem Switch und einer Menge von Modes. Wie im Abbildung 4.2 dargestellt ist, sind zusätzlich ein DataReader und ein DataWriter vorhanden. Diese beinhalten Komponenten, die die beiden Zwischenspeicher innerhalb der Schnittstelle realisieren. Generell sind verschiedene Implementierungen dieser Zwischenspeicher denkbar. Zum einen könnte man vorhandene Speicher der Zielplattform einsetzen zum anderen besteht in Hardware die Möglichkeit Speicher zu synthetisieren. Da die Daten, die im IFB in den einzelnen Zuständen

der Kommunikationsprotokolle vorkommen, stark variieren können, wurde die dedizierte Synthese des Speichers bevorzugt. Das Ergebnis dieser Synthese ergibt ein Register-File, welches für eine spezielle Ausprägung des IFBs exakt den benötigten Zwischenspeicher darstellt. Um das Register-File anzusprechen muss zusätzlich eine Speicherschnittstelle in Form eines endlichen Automaten bereitgestellt werden.

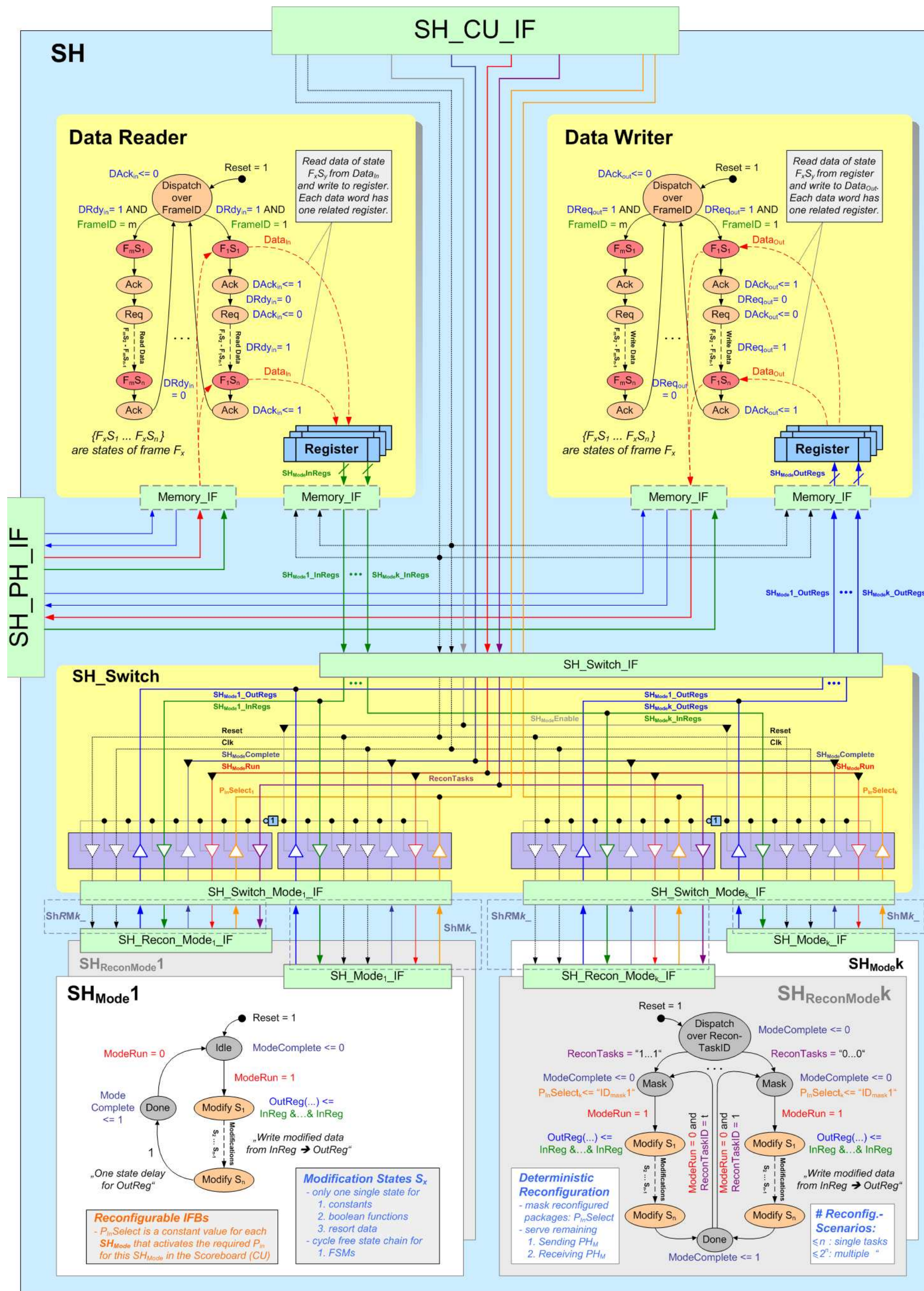

Abbildung 4.2.: Der Sequence Handler und seine Komponenten

4.1.3. Control Unit

Wie in Abbildung 3.3 gezeigt ist, wird das *Scoreboard* auf Gatterebene modelliert. Die Scheduler $Ctrl_{In}$ und $Ctrl_{Out}$ werden wie *Reconfiguration*-Einheit in Abbildung 3.4 durch endliche Zustandsautomaten implementiert.

Für die Kommunikation zwischen den Automaten des IFBs werden *Fully-Interlocked-Protokolle* verwendet. Diese sorgen dafür, dass sich die FSMs synchronisieren und keine Daten verloren gehen. Die für die Implementierung notwendigen Zustände werden bei der Erweiterung der eingehenden IFDs im ersten Syntheseschritt hinzugefügt.

4.1.4. Ergebnis der Analyse des IFB-Templates

Wie die Auswertung der Komponenten des IFBs zeigt, ist es erforderlich Zustandsautomaten sowie Gatterschaltungen in Form von *Verhaltensbeschreibungen* erzeugen zu können. Automaten werden für die PH_{Modes}, die SH_{Modes}, die Speicherschnittstelle des Registerfiles, die Reconfiguration Einheit und die beiden Scheduler Ctrl_{In} und Ctrl_{Out} generiert. Der Komponenten-basierte Aufbau der Module setzt die Erzeugung von *Strukturbeschreibungen* voraus. Auf diese Weise entsteht dann die Hierarchie der IFB-Makrostruktur.

Abbildung 4.3 veranschaulicht die Aufteilung zwischen Struktur- und Verhaltensbeschreibungen. Die grünen Komponenten sind dabei reine Strukturbeschreibungen, die blauen reine Verhaltensbeschreibungen. In den türkisen Komponenten werden beide Beschreibungsformen verwendet.

Im Weiteren wird davon ausgegangen, dass die Sprache VHDL und damit das Konzept der Struktur- und der Verhaltensbeschreibung bekannt sind. Dabei beschränkt sich der Codegenerator auf das synthesefähige Subset der Sprache VHDL [Pau04].

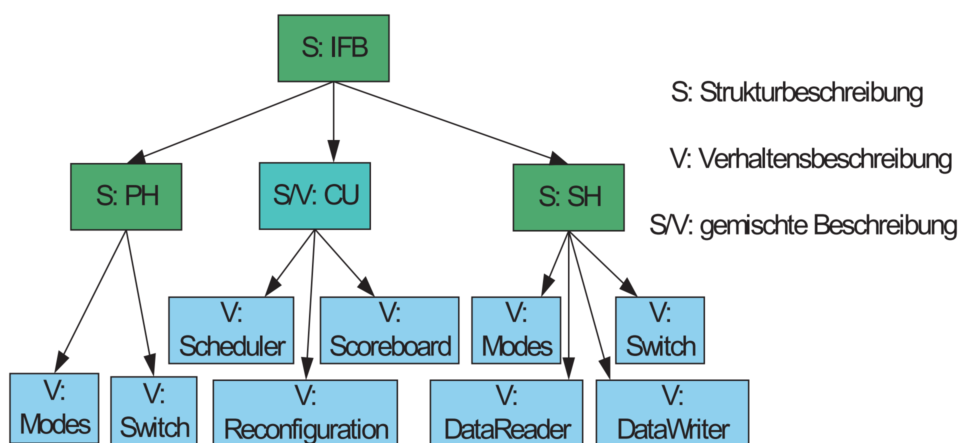

Abbildung 4.3.: Verhalten- und Strukturbeschreibung einer Architektur

4.2. Auswertung der Interface Descriptions (IFD)

Wie in Abbildung 3.5 veranschaulicht wurde, erzeugt der erste Syntheseschritt eine IFB-Description bestehend aus einer Menge von IFDs als Eingabe des Codegenerators. Wie in Kapitel 2 eingeführt besteht eine IFD aus den Elementen Interface (I), Protocol (P) und ProtocolMap (M). Um einen höheren Informationsgehalt in einer einzelnen IFD unterzubringen, existieren verschiedenartige Codierungen einer IFD. Die Codierung unterscheiden sich dadurch, dass nicht immer alle drei Elemente vorhanden sein müssen. Im Folgenden werden in der Beschreibung der möglichen Konstellationen fehlende Einträge durch ein "X" gekennzeichnet. Dabei sind die unten aufgeführten Konstellationen erlaubt:

- **I-P-M**: Die vollständige Codierung wird für die Beschreibung der Komponenten (z. B. der Modes) verwendet, die in eine Verhaltensbeschreibung von Zustandsautomaten übersetzt werden. Bei der Auswertung dieser Codierung wird aus dem Interface (I) die Entitybeschreibung sowie die Komponentendeklaration generiert. Aus dem Protocol (P) wird die Verhaltensbeschreibung erzeugt. Dabei handelt es sich immer um endliche Zustandsautomaten. Die ProtocolMap (M) wird genutzt, um die Signale, die in der Verhaltensbeschreibung benutzt werden, auf die Signale der Entitybeschreibung abzubilden. Die Schnittstellen der Tasks und Medien werden ebenfalls in dieser Codierung modelliert.
- **I-X-X**: Diese Codierung wird für die Erzeugung der Komponenten benutzt, für die ausschließlich eine Strukturbeschreibung erzeugt wird. Diese Komponenten sind in Abbildung 4.3 grün dargestellt (IFB, Sequence Handler und Protocol Handler). Aus dem Interface (I) wird auch hier wieder die Entitybeschreibung kreiert.
- **X-P-X**: Wie auch die *I-X-X* Codierung wird diese Art von IFD ausschließlich innerhalb des ersten Syntheseschrittes generiert. Dabei drückt diese Codierung innerhalb einer Strukturbeschreibung aus, wie Komponenten untereinander verbunden werden.

Abhängig von der jeweiligen Codierung und der Position des IFDs innerhalb der IFD-Description weiß der Codegenerator, welchen VHDL-Code er zu generieren hat.

Verhaltensbeschreibungen auf Gatterebene können nicht durch IFDs modelliert werden. Deshalb synthetisiert der Codegenerator selbständig die Komponenten des IFBs, die durch Gatterschaltungen implementiert werden. Aus diesem Grund werden die Switches und das Scoreboard im ersten Syntheseschritt nicht berücksichtigt. Dieses Vorgehen beruht darauf, dass die drei oben genannten IFD-Codierungen in ähnlicher Weise für eine Softwarerealisierung genutzt werden könnten, eine Gatterschaltung aber keine Abbildung finden würde. Da die IFB-Description unabhängig von der Art der Realisierung des IFBs ist, müssen alle Teile des Synthesealgorithmus, die zwischen einer Realisierung in Hardware und Software unterscheiden, als zweiter Syntheseschritt direkt in die Codegenerierung integriert werden.

4.3. Aufbau des Codegenerators

Für die Implementierung des Codegenerators wird das Konzept des *Frame Processings* angewendet. Abbildung 4.4 zeigt die für diese Arbeit angepasste Form des Frame Processings aus Abbildung 2.3. Das Metamodell wird hier als *VHDL Gen* bezeichnet. Die Verwaltung der Frames geschieht durch die so genannte *VHDL Factory*.

Frame Processing ist für die Generierung von VHDL-Code besonders geeignet, da VHDL eine Komponenten-orientierte Sprache ist. Das führt dazu, dass einzelne Teile von VHDL einfach auf Frames abgebildet werden können. Folglich werden die *Frames* dazu verwendet, Elemente des IFB-Templates als VHDL-Pattern zu erzeugen. Nachfolgend wird die Funktionalität von *VHDL Gen* und *VHDL Factory* näher erläutert.

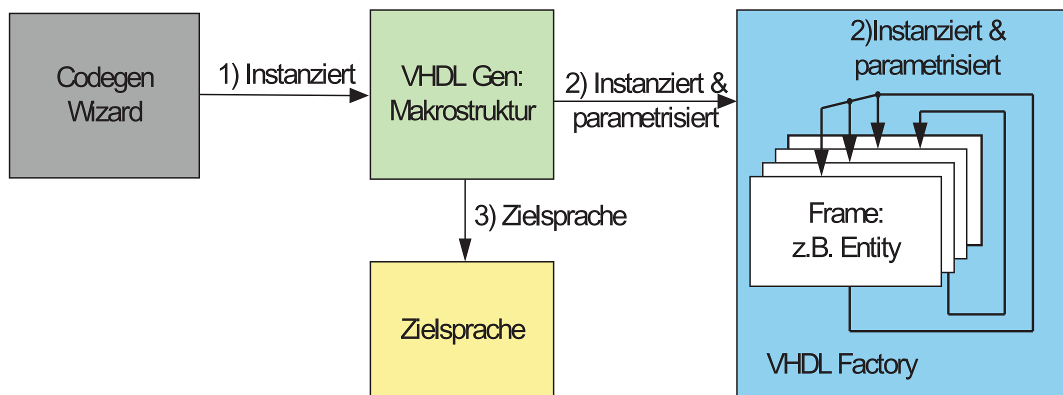

Abbildung 4.4.: VHDL-Codegenerierung durch Frame Processing

4.3.1. VHDL Gen

Wie im Abbildung 4.4 zu sehen ist, wird die IFB-Makrostruktur in *VHDL Gen* als Metamodell verwendet. Zur Erzeugung des VHDL-Codes durchläuft *VHDL Gen* in einem rekursiven Abstieg dieses Metamodell und nutzt dabei die *VHDL Factory*, um die abstrakten Beschreibungselemente der IFDs in VHDL-Code umzusetzen.

VHDL Gen berücksichtigt sowohl die Hierarchie der Komponenten im IFB-Template als auch die interne Struktur innerhalb einer VHDL-Beschreibung, wie z. B. den Aufbau der Entity- oder Architekturbeschreibung. Der Ausgangspunkt des rekursiven Abstiegs beginnt auf der Ebene des Interface Blocks.

Der erzeugte VHDL-Code wird zentral in *VHDL Gen* verwaltet. Dazu übergeben die Frames der *VHDL Factory* die erzeugten Codefragmente an *VHDL Gen*. Nach erfolgter Codegenerierung kann der generierte Code dann von hier aus z. B. als VHDL-Dateien exportiert werden. Eine Implementierung dieser Technik wird in Kapitel 5 vorgestellt.

4.3.2. VHDL Factory

Die Generierung des eigentlichen VHDL-Codes erfolgt durch das Instanzieren von Frames in der *VHDL Factory*. Ein Frame besteht aus Slots. Ein Slot kann entweder aus einem weiteren Frame oder einem Codeschnipsel bestehen. Dabei stellt jeder Schnipsel ein parametrisiertes Stück VHDL-Code dar. Bei der Instanzierung eines Frames erhalten die parametrisierten Codeschnipsel ihre endgültige Ausprägung. Ein Beispiel dafür ist der Bezeichner einer Entity. Im Codeschnipsel ist dieser als Parameter (hier: entity_Name) enthalten und wird bei der Erzeugung durch einen konkreten Wert ersetzt.

```
entity entity_Name is
  port( ... );
end [entity_Name];
```

Im folgenden Abschnitt wird die Codegenerierung der wichtigsten Frames exemplarisch beschrieben.

4.4. Erzeugung der VHDL-Pattern

Eine VHDL-Beschreibung besteht aus mehreren Teilen: einer Schnittstellenbeschreibung (*Entity*), einer oder mehreren Verhaltens- oder Strukturbeschreibungen (*Architecture*) und den Konfigurationen (*Configurations*). Für den Interface Block werden keine Konfigurationen benötigt, da zu jeder Entity genau eine Architecture erzeugt wird und somit eine eindeutige Bindung besteht.

Zunächst wird die Erzeugung von Entities erläutert. Anschließend wird die Erzeugung der Architectures inklusive der Protokollzustandsmaschinen (FSM) beschrieben. Abschließend wird die Synthese und Codegenerierung der Register-Files betrachtet, die für die interne Kommunikation im IFB notwendig sind.

Der Codegenerator erhält die IFB-Description als Datenstruktur im Hauptspeicher als Eingabe aus dem ersten Syntheseschritt. Um die darin vorkommenden IFDs zu visualisieren, werden diese nachfolgend in ihrem Austausch-Format als XML (Extensible Markup Language) repräsentiert.

4.5. Generierung einer Entity

VHDL Komponenten kommunizieren über ihre *Entity* miteinander. Die Kommunikationskanäle dieser Schnittstellenbeschreibung werden als *Ports* bezeichnet. Für die Erzeugung eines Ports müssen die Attribute: Name, Datentyp, Signalflussrichtung und Signalbreite bekannt sein. Eine *Entity* ist folgendermaßen aufgebaut:

```
Entity entity_Name is
Port(
      port_name_1 : port_direction_1 port_type_1;
      port_name_2 : port_direction_2 port_type_2;
          .
          .
          .
      port_name_n : port_direction_n port_type_n
   );
end [entity_name];
```

Das unten angegebene XML Listing 4.1 zeigt beispielhaft die Schnittstelle einer Interface Description (IFD) bestehend aus einem Port mit der Bitbreite eins. Dabei handelt es sich um das globale IFB Taktsignal.

XML Listing 4.1: IFD Interface

```
<Interface>
   <Identification>
      <name>IFB_IF</Name>
      <Description>IFB_System_Interface</Description>
   </Identification>
   <PhysicalStructure>
      <PortList>
         <Port>
            <Identification>
               <Name>IFB_Clk</Name>
               <ID>1</ID>
               <Description>Global_IFB_Clock</Description>
            </Identification>
            <PinList>
               <Pin>
                  <Identification>
                     <Name>IFB_Clk</Name>
                     <ID>1</ID>
                     <UserID>1</UserID>
                  </Identification>
                  <Characterization>
                     <Direction>Input</Direction>
                     <Type>Std_Logic</Type>
                  </Characterization>
               </Pin>
            </PinList>
         </Port>
      </PortList>
    </PhysicalStructure>
</Interface>
```

VHDL Listing 4.1 zeigt den Code, der aus dem im XML Listing 4.1 dargestellten Interface generiert wird. Der Bezeichner der Entity wurde dabei von dem Namen des IFDs (hier nicht gezeigt) abgeleitet. Ein Interface der IFD beinhaltet eine "PortList", die wiederum aus *Ports* besteht. Die für die Codegenerierung interessanten Attribute eines Ports sind: Name, Direction, Type, Description und *PinList*. Eine PinList besteht aus *Pins*, aus deren Anzahl die Breite des Ports festgestellt wird. Um die Verständlichkeit des VHDL-Codes zu erhöhen werden die Signale einer "PortList" mit einem Kommentar versehen, der sich aus der *Description* des Interfaces ableitet. Im VHDL Listing 4.1 handelt es sich zum Beispiel um ein Interface namens "IFB_System_Interface".

VHDL Listing 4.1: Generierter VHDL-Code einer Entity

```
Entity IFB is
Port(
   ---------------------------------------------------------
   -- IFSInterface: IFB_System_Interface                  --
   ---------------------------------------------------------
      IFB_Clk : in std_logic
   );
end IFB;
```

4.6. Generierung einer Architecture

Neben der Entity ist die Architecture der zweite Teil einer jeden VHDL-Datei. Sie beschreibt die Funktionalität und somit die interne Realisierung einer Komponente. Wie in Abbildung 4.3 gezeigt wurde, sind Verhaltens- und Strukturbeschreibungen erforderlich. Beiden Beschreibungsarten liegt das gleiche Skelett einer Architecture zugrunde:

```
architecture architecture_name of entity_name is
  [arch_declarative_part]
begin
  [arch_statement_part]
end [architecture_name];
```

Die Architecture untergliedert sich in den Deklarationsteil *arch_ declarative_ part* vor dem "begin" und die Beschreibung des Verhaltens in *arch_ statement_ part*. Im Deklarationsteil werden unter Anderem lokale Typen, lokale Signale und Komponenten deklariert. Die Erzeugung der Komponentendeklaration basiert auf dem gleichen Vorgehen wie die Erzeugung der Entity und nutzt dazu teilweise sogar die gleichen Frames. Die eigentliche Modulbeschreibung erfolgt daran anschließend im "arch_statement_part". In Folgenden, wird die Generierung einer Verhaltensbeschreibung vorgestellt.

4.6.1. Generierung der Verhaltensbeschreibung

Verhaltensbeschreibungen werden in die Kategorien nebenläufige- bzw. sequentielle Verhaltensbeschreibung unterteilt. Grundlage einer sequentiellen Verhaltensbeschreibung ist die Prozess-Umgebung. Ein Prozess wird innerhalb einer Architecture als nebenläufige Anweisung betrachtet. Verhaltensbeschreibungen im IFB kommen entweder als Gatterschaltung oder als Moore-Automaten vor. Beide Arten von Verhaltensbeschreibung werden als Prozesse implementiert. Zunächst wird die Generierung der Moore-Automaten näher betrachtet.

Generierung einer FSM (Moore-Automat)

XML Listing 4.2: IFD Protocol

```
<Protocol>
    <Identification>
        <name>Data_IF_GenPr</Name>
        <ID>1</ID>
        <Description>Generated Protocol from Interface Data_IF (ID: 1)</Description>
    </Identification>
    <Version>...</Version>
    <ProtocolPinList>
        <ProtocolPin>
            <Identification>
                <name>TxD</Name>
                <ID>1</ID>
                <Description>Transmit Data</Description>
            </Identification>
            <Characterization>
                <Direction>Output</Direction>
                <Type>Std_Logic</Type>
                <Behavior>Data+Control</Behavior>
            </Characterization>
        </ProtocolPin>
    </ProtocolPinList>
    <StateList>...</StateList>
    <ReferenceSignals>...</ReferenceSignals>
</Protocol>
```

XML Listing 4.2 zeigt die XML Repräsentation eines Protokolls. Ein *Protocol* beinhaltet eine *ProtocolPinList*, eine *StateList* und eine Liste von *ReferenceSignals*. *ProtocolPins* sind Signale, deren Werte in den Zuständen der StateList festgelegt werden. Wie VHDL Listing 4.2 zeigt, werden ProtocolPins in der Architecture als "lokale" Signale deklariert und durch die ProtocolMap der IFD auf die Signale der Entity abgebildet. Auf die Bedeutung der ReferenceSignals als Zeitbasis für die FSMs wird später eingegangen.

VHDL Listing 4.2: Signaldeklaration eines ProtocolPins

```
Architecture PH_Mode_0_Behavior of PH_Mode_0 is
----------------------------------------------------------
-- Definition of Local Signals                          --
----------------------------------------------------------
 signal TxD_loc : std_logic;

begin
----------------------------------------------------------
-- Mapping: Entity <=> Local Signals                    --
----------------------------------------------------------
 PhM0_TxD <= TxD_loc;
 ...
```

Zunächst folgt die Beschreibung der StateList:

XML Listing 4.3: State aus der StateList in Protocol

```
<state>
   < Identification >
      <Name>Idle</Name>
      <ID>1</ID>
      <Description>Wait for data to send</Description>
   </Identification>
   <TransitionList>
      <Transition>
         < Identification >
            <name>SetStartBit</Name>
            <ID>1</ID>
            <Description>Transition to state 2.</Description>
         </Identification>
         <NextStateID>2</NextStateID>
         <TransitionConditionList>
            <TransitionCondition>
               <Operator>AND</Operator>
               <Trigger>
                  <SignalSource>ProtocolPin</SignalSource>
                  <TriggeredSignalID>1</TriggeredSignalID>
                  <ExpectedValue>Falling Edge</ExpectedValue>
               </Trigger>
            </TransitionCondition>
         </TransitionConditionList>
      </Transition>
   </TransitionList>
   <MooreOutputList>
```

```
        <AutomataOutput>
            <ProtocolPinID>1</ProtocolPinID>
            <Value>High Value</Value>
            <UseCase>Outgoing Control</UseCase>
        </AutomataOutput>
    </MooreOutputList>
</state>
```

Wie XML Listing 4.3 veranschaulicht, wird ein Zustand (*State*) des Moore-Automaten durch eine *Identification*, seine Zustandsübergänge (*TransitionList*) und eine Menge von Ausgaben (*MooreOutputList*) charakterisiert. Dabei besteht jeder Zustandsübergang aus einer Menge von Zustandsübergangsbedingungen (*TransitionConditionList*). Aus diesen Attributen wird die Deklaration der Zustände im Deklarationsteil der Architektur erzeugt, wie in VHDL Listing 4.3 dargestellt.

VHDL Listing 4.3: Zustandsdeklaration des Automaten

```
----------------------------------------------------------
-- Declaration of State_Type Definition                --
----------------------------------------------------------
type StateType is (
                    Idle,      -- Wait for data to send
                    Start,     -- Sending start bit
                    ...
                   );
signal CurrentState, NextState : StateType;
```

VHDL Listing 4.4 zeigt Ausschnitte des zugehörigen Moore-Automaten:

VHDL Listing 4.4: Prozess des Automaten

```
----------------------------------------------------------
-- FSM Processes                                        --
----------------------------------------------------------
PH_Mode_0_TransitionAndOutput: process (CurrentState) begin

   case CurrentState is
      when Idle =>
         TxD_loc <= '1';
         if (ModeRun = '1') then
            NextState <= Start;
         else
            NextState <= Idle;
         end if;
   ...
```

```
PH_Mode_0_Synchronize: process (PhM0_IFB_Clk, PhM0_IFB_Reset) begin
   if (PhM0_IFB_Reset = '1') then
      CurrentState <= Idle;       -- Reset to Initial State
   elsif (PhM0_IFB_Clk'event and PhM0_IFB_Clk = '1') then
      CurrentState <= NextState; -- Advance Current State
   end if;
end process;
```

Für jeden zu erzeugenden Zustandsautomaten werden insgesamt drei Prozesse angelegt. Dies geschieht, um die VHDL-Beschreibungen für die Synthese möglichst einfach zu halten [Dou96] und die Lesbarkeit des Codes zu erhöhen. Der erste Prozess ist oben in VHDL Listing 4.4 dargestellt und realisiert die Zustandsübergangsfunktion sowie die Ausgabefunktion für die Signale, denen in jedem Zustand eine Ausgabe zugewiesen wird. Diese Ausgabefunktion kann in der Low-Level Synthese später durch eine einfache Schaltlogik implementiert werden. Der zweite, hier nicht dargestellte Prozess, behandelt solche Ausgaben, die nur in einigen Zuständen zugewiesen werden. Für diese Signale werden in der Low-Level Synthese zusätzlich Speicherelemente synthetisiert. Der dritte Prozess realisiert die Zustandsfortschaltung, wie in VHDL Listing 4.4 dargestellt. Dabei werden ausschließlich synchrone Automaten mit asynchronem Reset erzeugt.

ReferenceSignals sind ein wichtiger Aspekt der Protokollbeschreibung. Sie erlauben benutzerspezifische Zeitangeben in den Zustandübergangsbedingungen. Ein ReferenceSignal kann ein *GlobalDate*, ein *TimerOrDeadline* oder eine *ReferenceClock* sein. *GlobalDates* werden benutzt, um periodisch auftretende Signale zu beschreiben. *ReferenceClocks* sind künstliche Clocks, die aus einer bereits existierenden Clock abgeleitet werden. Ziel dabei ist es eine neue Clock mit passender Taktrate zu generieren. Ein *TimerOrDeadline* kann entweder als Timer oder als Deadline eingesetzt werden [Mic05].

Alle ReferenceSignals werden auf Basis von Clock-Teilern als Zähler implementiert. Die GlobalDates besitzen zusätzlich noch eine Look-Up-Table, in der die Zweitwerte der einzelnen Signale gespeichert sind. Für alle GlobalDates wird ein Deklarationsteil und ein separater Prozess angelegt.

Generierung einer Verhaltensbeschreibung auf Gatterebene

Wie bereits diskutiert wurde, werden Verhaltensbeschreibungen auf Gatterebene nicht durch IFDs modelliert. Für die Generierung dieser Gatterschaltungen existieren spezielle Frames, die aufgrund eines gegebenen Algorithmus Code generieren.

Im Switch zum Beispiel hängt der generierte VHDL-Code von der Anzahl der Modes und deren Leitungen ab. Der Algorithmus zur Generierung des Scoreboards analysiert das IFD-Mapping (siehe Kapitel 2). Dabei werden unter Anderem die Anzahl der Sh_{Modes} und die zu transformierenden Datenpakete ausgewertet.

4.6.2. Generierung der Strukturbeschreibung

Eine Strukturbeschreibung besteht aus der Deklaration, der Instanzierung und der Verdrahtung von bestehenden Komponenten. Die Information, welche Komponenten zu einer Architecture gehören findet sich in der Struktur der IFB-Description.

So existiert z. B. für jeden Mode innerhalb eines Handlers eine I-P-M codierte IFD. Da der Switch als Gatterschaltung angelegt wird, liegt dessen IFD in der Codierung I-X-X vor. Um auszudrücken, welche Signale von Mode und Switch auf der Ebene des Handlers miteinander zu verbinden sind, wird eine zusätzliche IFD in der Codierung X-P-X hinzugefügt. VHDL Listing 4.5 zeigt einen Teil der generierten Strukturbeschreibung.

VHDL Listing 4.5: VHDL Strukturbeschreibung

```
-- Declaration of Components
Component PH_Switch
Port(
      PhM0_IFB_Clk   : out std_logic;
      PhM0_IFB_Reset : out std_logic;
      ... );
end Component;  --<PH_Switch>--

Component PH_Mode_0
Port(
      PhM0_IFB_Clk   : in std_logic;
      PhM0_IFB_Reset : in std_logic;
      ...  );
end Component;  --<PH_Mode_0>--

-- Definition of Local Signals
signal PhM0_IFB_Clk_loc   : std_logic;
signal PhM0_IFB_Reset_loc : std_logic;

begin
PH_Mode_0_Inst: PH_Mode_0
Port Map(
         PhM0_IFB_Clk   => PhM0_IFB_Clk_loc,
         PhM0_IFB_Reset => PhM0_IFB_Reset_loc,
        ...  );

PH_Switch_Inst: PH_Switch
Port Map(
         PhM0_IFB_Clk   => PhM0_IFB_Clk_loc,
         PhM0_IFB_Reset => PhM0_IFB_Reset_loc,
        ...  );
```

4.7. Synthese der Register-File

Um die Synthese der Register-Files zu erklären, werden zunächst die Begriffe *Frame* und *Package* aus dem IFD-Mapping eingeführt. "Frame" ist dabei nicht mit dem Begriff aus dem Frame-Processing zu verwechseln.

4.7.1. Frames und Packages

Im ersten Syntheseschritt werden die Protokollzustandsautomaten in den IFDs in Grundblöcke unterteilt. Die Grundblöcke repräsentieren die möglichen Paket-Typen eines Protokolls. Das serielle Protokoll z. B. kennt nur einen Paket-Typ, den es immer wiederholt und besteht daher nur aus einem Grundblock. Informationen zu Grundblöcken sind in [Tob05] angegeben.

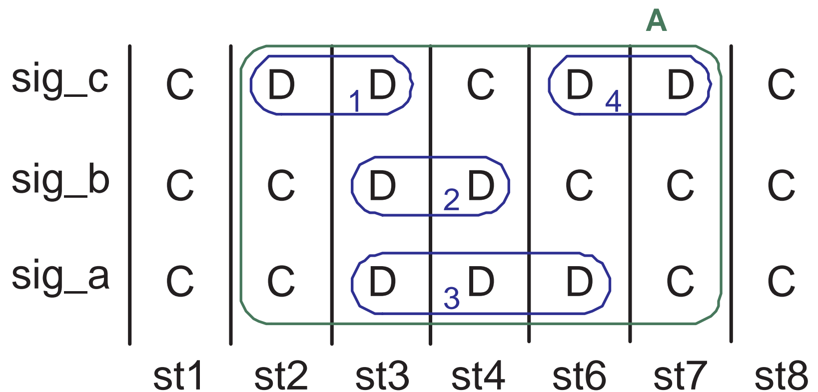

Abbildung 4.5.: Protocol-Matrix

Abbildung 4.5 illustriert ein solchen Grundblock. Die Spalten repräsentieren die Zustände des zugehörigen Protokollautomaten. Jede Zeile steht für einen ProtocolPin. Die mit **C** gekennzeichneten Einträge des Grundblocks sind Control Bits und daher für die Übertragung im IFB irrelevant. Bits, die als **D** markiert sind und im IFD-Mapping ausgewählt wurden, sind durch den IFB umzusetzen. Dazu wird genau für diese Bits ein spezifisches Register-File erzeugt.

Für die Synthese des Register-Files werden nun Frames innerhalb der Grundblöcke gebildet. Dazu werden zuerst zeilenweise aufeinanderfolgende Datenbits zu Datenpaketen, den sogenannten *Packages*, zusammengefasst. Spaltenweise überlappende Datenpakete werden danach zu *Frames* verschmolzen [Die05]. In Abbildung 4.5 stellt das in grün eingekreiste Rechteck einen *Frame* dar. Die blau umrandeten Rechtecke repräsentieren die *Packages*. Allgemein kann ein Grundblock aus mehreren Frames und dieser wiederum aus mehreren Packages bestehen.

4.7.2. Register-Files

Das Register-File wird als VHDL-Package generiert und als *Library* in die übrigen VHDL-Dateien eingebunden. Das VHDL-Package wird als eine hierarchische Struktur aus VHDL-Records angelegt. Aus dem in Abbildung 4.5 dargestellten Frame, wird das folgende Register-File synthetisiert:

```
package RegFile is

type PACKAGE_1 is record
   sig_a  : std_logic_Vector(2 downto 0);
end record;

type PACKAGE_2 is record
   sig_b  : std_logic_Vector(1 downto 0);
end record;

type PACKAGE_3 is record
   sig_c  : std_logic_Vector(1 downto 0);
end record;

type PACKAGE_4 is record
   sig_c  : std_logic_Vector(1 downto 0);
end record;

type FRAMEINST_1 is record
   Pck1 : PACKAGE_1;
   Pck2 : PACKAGE_2;
   Pck3 : PACKAGE_3;
   Pck4 : PACKAGE_4;
end record;

type FRAME_1 is record
   FI1 : FRAMEINST_1;
end record;

end RegFile;
```

Durch die vorgestellte Codegenerierung entstehen eine Menge synthesefähiger VHDL-Beschreibungen. Diese Dateien können anschließend im letzten Schritt des IFS-Flows mit weiteren Synthesewerkzeugen, die VHDL als Eingabesprache akzeptieren, weiterverarbeitet werden.

5. Implementierung des Codegenerators

In diesem Kapitel wird gezeigt, wie das in Kapitel 4 erläuterte Konzept in Java implementiert wurde. Im Folgenden werden Klassendiagramme für die zwei Hauptbereiche des Codegenerators vorgestellt. Dabei wird lediglich auf die wichtigsten Klassen eingegangen. Anschließend werden einige spezielle Datenstrukturen vorgestellt, die eingeführt wurden um die Verwaltung der erzeugten Codeschnipsel zu implementieren. Zum Schluss wird die grafische Oberfläche (GUI) des Codegenerators (*CodegenerationWizard*) präsentiert. Zunächst wird das Package-Diagramm des Codegenerators vorgestellt, um einen Überblick zu schaffen.

5.1. Package-Diagramm des Codegenerators

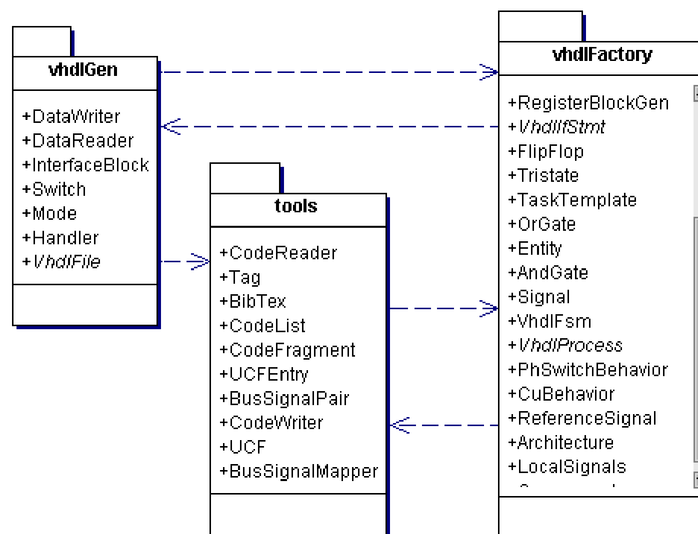

Abbildung 5.1.: Codegenerator

Wie in Abbildung 5.1 veranschaulicht wird, werden für die Implementierung des Codegenerators die Packages *vhdlGen vhdlFactory* und *tools* genutzt. Im Package tools, befinden sich Klassen wie z. B. *CodeWriter*, *CodeList*, *CodeFragment* und *Tag*.

der *CodeWriter* wird genutzt, um den generierten Code in eine Datei zu schreiben. Auf die Klassen CodeList, CodeFragment und Tag wird später eingegangen.

5.2. Klassendiagramme

5.2.1. vhdlGen

Abbildung 5.2 zeigt das Package vhdlGen. Für jede Komponente der IFB-Makrostruktur existiert eine Klasse in diesem Package. Neben diesen Klassen beinhaltet vhdlGen eine abstrakte Klasse namens *VhdlFile*, die den Aufbau der zu generierenden VHDL-Beschreibungen durch Entity und Architecture festlegt. Die Control Unit, der Sequence Handler und der Protocol Handler werden durch die Klasse *Handler* beschrieben. Die Klasse *Mode* repräsentiert sowohl die Modes der Protocol- und Sequence Handler als auch die Module der Control Unit.

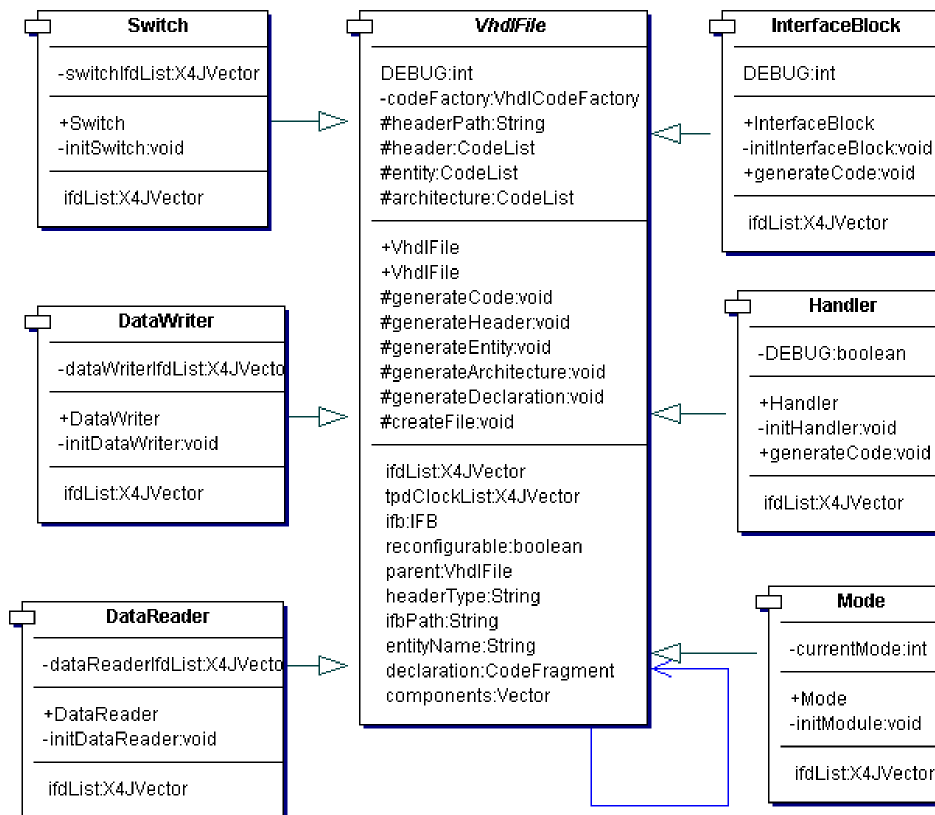

Abbildung 5.2.: Package VhdlGen

5.2.2. vhdlFactory

VhdlFactory beinhaltet Klassen, die die Frames repräsentieren. Diese Klassen sind für die Erzeugung der VHDL-Pattern zuständig. Wie in Abbildung 5.3 visualisiert wird, beinhaltet vhdlFactory Klassen wie Entity, Architecture und Component. Die Klasse Entity wird für die Generierung der Entitybeschreibung eingesetzt. Die Klasse Component wird für die Erzeugung der Komponentendeklaration benutzt. Beide Klassen erben von der abstrakten Klasse VhdlPort aufgrund ihres ähnlichen Aufbaus. Die Klasse Architecture wird für die Erzeugung der Architekturbeschreibung eingesetzt. Dabei benutzt sie Klassen wie Component und Process, um ihre Pattern zu generieren.

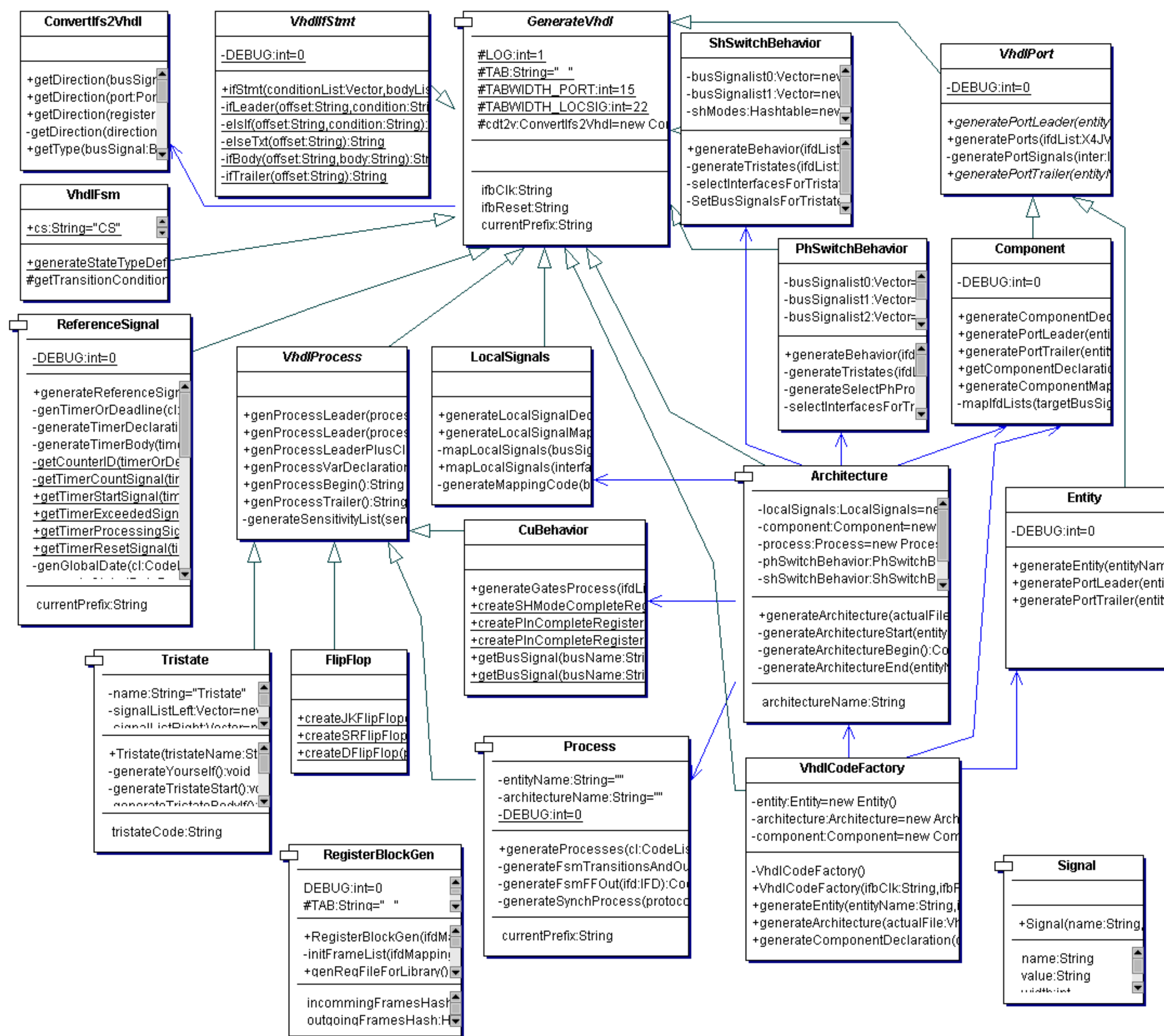

Abbildung 5.3.: Package VhdlFactory

5.3. Verwendete Datenstrukturen

5.3.1. CodeList, CodeFragment, Tag

Durch die Klassen der CodeFactory werden bei der Codegenerierung die VHDL-Codeschnipsel erzeugt. Die Datenstruktur, die den erzeugten Code beinhaltet wird als *CodeFragment* bezeichnet. Um VHDL-Beschreibungen aus mehreren CodeFragments zu erzeugen, müssen diese in der richtigen Reihenfolge zusammengesetzt werden. Aus diesem Grund wurden die Datenstrukturen *CodeList* und *Tag* eingeführt. Die drei Datenstrukturen werden von vhdlGen zur Verwaltung des Codes während der Codegenerierung genutzt.

Eine *CodeList* ist eine verkettete Liste die verschachtelt werden kann. Die Blattknoten der Liste bestehen aus den CodeFragments. Ein *CodeFragment* kann in der *CodeList* durch seinen *Tag* eindeutig identifiziert werden. Die Klassen der vhdlFactory nutzen die Tags, um die erzeugten CodeFragments bzw. CodeLists in die CodeList von vhdlGen einzufügen.

5.4. Automatisierung: CodegenerationWizard

Im Rahmen dieser Arbeit wurde für den Codegenerator eine grafische Oberfläche namens *CodegenerationWizard* implementiert. Der CodegenerationWizard besteht aus drei Schritten (in den Abbildungen 5.4, 5.5 und 5.6 dargestellt).

1. Zuerst wird die abstrakte Zwischenrepräsentation des zu generierenden IFBs gewählt.
2. Im zweiten Schritt wird ein Ordner gewählt, wohin die generierten VHDL-Dateien abgelegt werden sollen.
3. Im dritten Schritt geschieht die eigentliche Codegenerierung.

Die Schritte 1 und 2 finden interaktiv statt. Die Codegenerierung erfolgt automatisch.

Abbildung 5.7 zeigt einen Ausschnitt aus einer generierten VHDL-Beschreibung. Dieser Ausschnitt zeigt die Schnittstellenbeschreibung eines Protocol Handler Modes. Das Programm, welches hier zum Anzeigen des Codes verwendet wurde ist ISE 6.1 der Firma Xilinx.

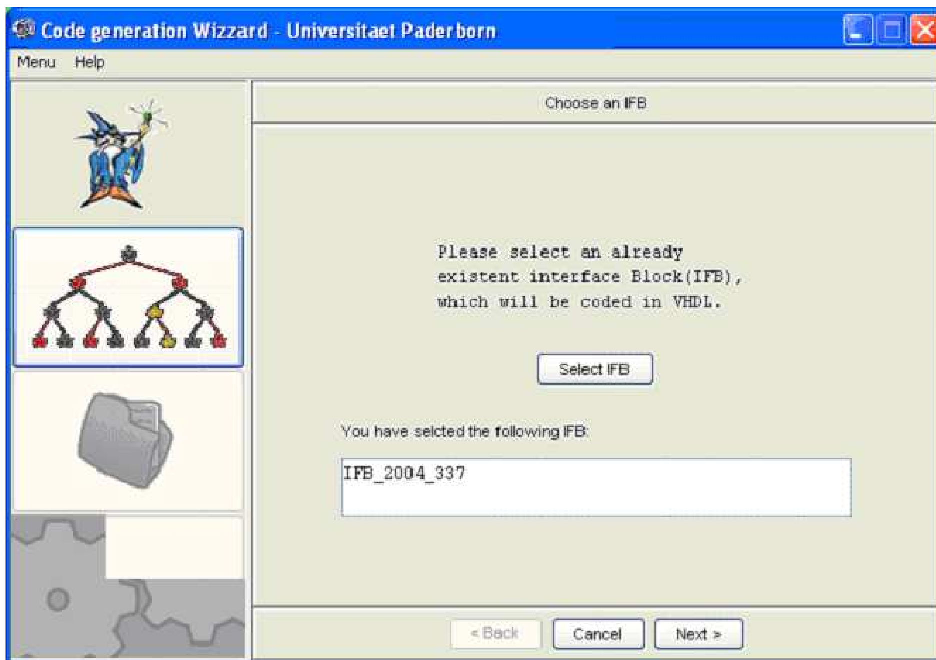

Abbildung 5.4.: CodegenerationWizard, Schritt 1 - Auswahl des IFBs

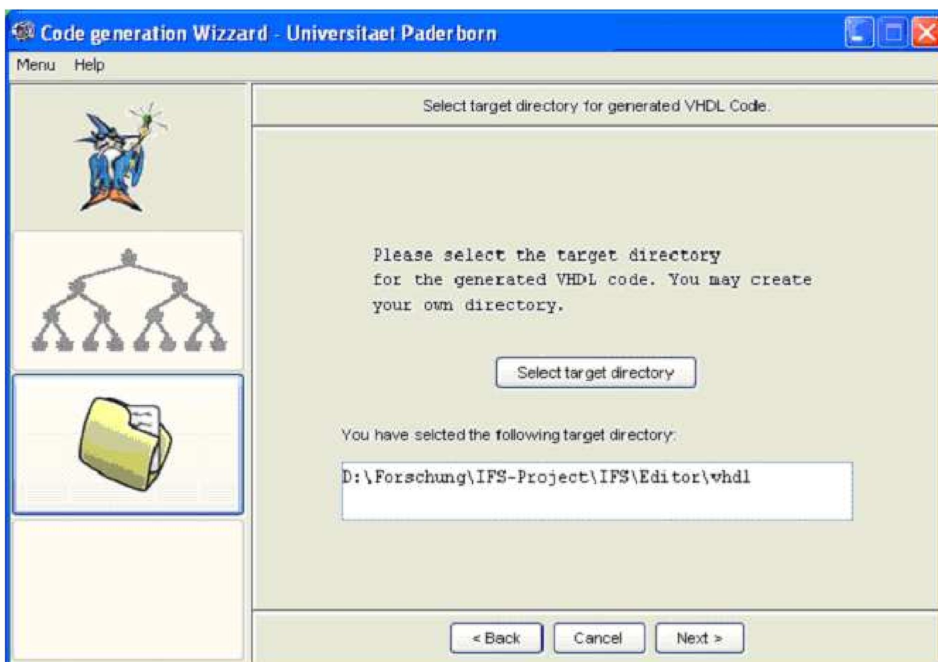

Abbildung 5.5.: CodegenerationWizard, Schritt 2 - Auswahl des VHDL Zielordners

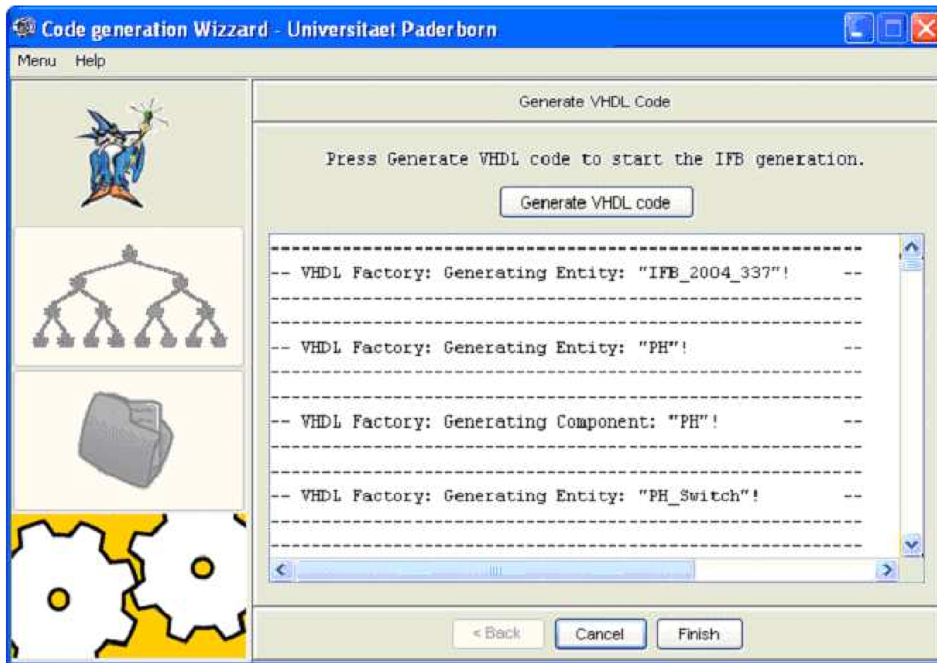

Abbildung 5.6.: CodegenerationWizard, Schritt 3 - Generierung des VHDL-Codes

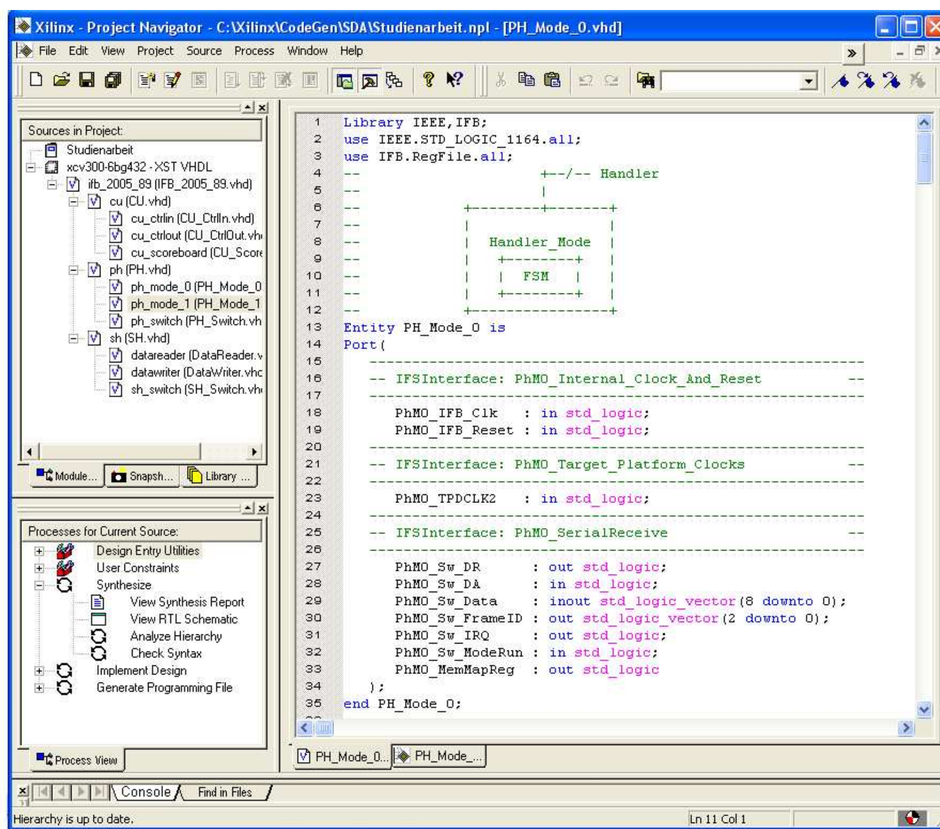

Abbildung 5.7.: Xilinx ISE 6.1 - Ausschnitt aus einer generierten VHDL-Beschreibung

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

Im Rahmen dieser Studienarbeit wurde ein VHDL-Codegenerator implementiert und als Teil des IFS-Flows in den IFS-Editor integriert.

Zunächst wurden einige existierenden Codegenerierungstechniken präsentiert. Danach wurde der IFS-Flow und der IFB eingeführt. Weiterhin wurden die Konzepte der IFB-Rekonfiguration vorgestellt. Ein wichtiger Teil dieser Arbeit bestand darin, den Inhalt und das Format der Ausgabe des ersten Syntheseschrittes mitzugestalten, da diese gleichzeitig als Eingabe für die Codegenerierung dient. Dabei wurden die Aspekte der Synthese, die von der Zielsprache VHDL abhängen (wie z. B. die Gatterschaltungen des Scoreboards und des Switches) direkt in die Codegenerierung integriert.

Im weiteren Verlauf der Arbeit zeigte die Analyse des IFB-Templates welcher VHDL zu erzeugen ist. Basierend darauf wurde die Auswertung der Eingabe des Codegenerators erläutert. Anschließend wurde der Aufbau des Codegenerators vorgestellt. Bei der Implementierung des Codegenerators wurde die Technik des Frame Processings angewendet. Dabei wurde verdeutlicht, dass die VHDL-Codegenerierung auf den Interface Block spezialisiert ist. Darauffolgend wurden die Konzepte zur Generierung der VHDL-Pattern erläutert. Dabei wurde exemplarisch auf die Generierung der wichtigsten Frames eingegangen. Zum Schluss wurde durch die Analyse des Klassendiagramms die Implementierung des Codegenerators erläutert.

Der vorgestellte Codegenerator ist in der Lage synthetisierbaren VHDL-Code sowohl für die rekonfigurierbare als auch für die nicht-rekonfigurierbare Version des IFBs zu generieren. Durch den Einsatz dieses Codegenerators, wurde ein durchgängiger Entwurfsablauf von der Modellierung bis zur Erzeugung des endgültigen IFB-Codes ermöglicht.

6.2. Ausblick

Im Rahmen dieser Arbeit wurde ein Codegenerator implementiert, der synthesefähiges VHDL erzeugt. Alternativ könnte ein Codegenerator entwickelt werden, der den IFB in einer anderen Hardware-Beschreibung-Sprache, wie zum Beispiel HandelC generiert. Ebenso wäre es sehr interessant den IFB als Softwarerealisierung für einen Prozessor als Zielplattform erzeugen zu können.

A. Anhang

A.1. SynthesisWizard

Für den ersten Syntheseschritt des IFS-Flows, wurde ein Wizard namens *Synthesis-Wizard* implementiert. Der *SynthesisWizard* leitet den Benutzer durch fünf Schritten durch.

1. Im ersten Schritt (in Abbildung A.1 dargestellt) werden die Schnittstellen der Tasks bzw. Medien gewählt, die zu verbinden sind .
2. Im zweiten Schritt (in Abbildung A.2 dargestellt) wählt der Benutzer die Zielplattform auf die, der nach der Codegenerierung generierte IFB geladen wird.
3. Im dritten Schritt (in Abbildung A.3 dargestellt) wird das IFDMapping (siehe Kapitel 2) gestartet um die Datenabbildung zu definieren. In diesem Schritt wird auch festgestellt ob der IFB rekonfigurierbar sein soll oder nicht.
4. Im vierten Schritt (in Abbildung A.4 dargestellt) wird der IFB in ein zwischen-Format generiert.
5. Im fünften Schritt (in Abbildung A.5 dargestellt) werden die Clock und Reset Signale gewählt und anschließend wird der IFB verbunden.

Abbildung A.6 kann z. B. als Ergebnis eines ersten Syntheseschrittes betrachtet werden. Links ist ein nicht-rekonfigurierbarer IFB zu sehen und rechts ein rekonfigurierbarer IFB mit einer Rekonfiguration Einheit, Protocol Handler Modes und Sequence Handler Modes. Die IFBs sind mit den Tasks verbunden.

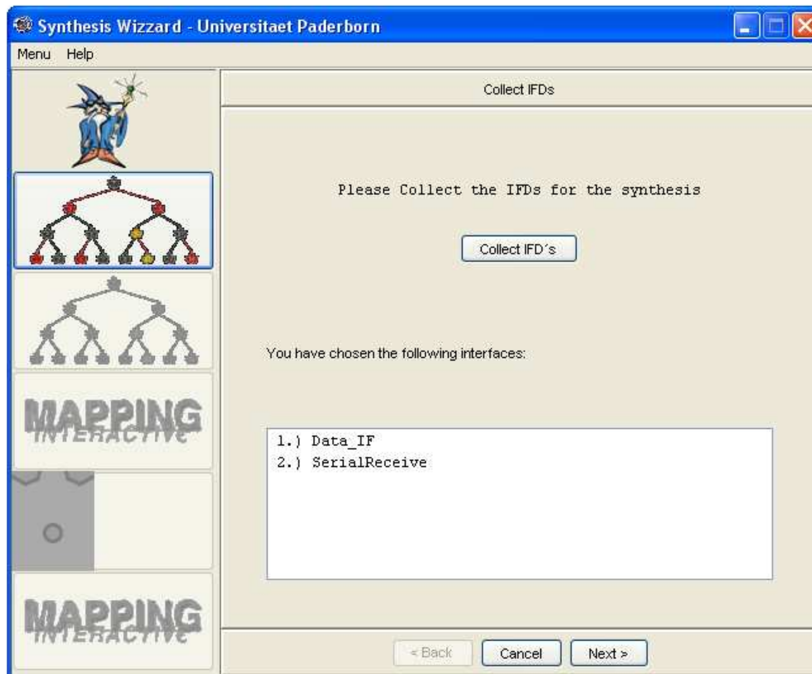

Abbildung A.1.: Synthesis Wizard, Schritt 1 - Auswahl der zu verbindenden Schnittstellen

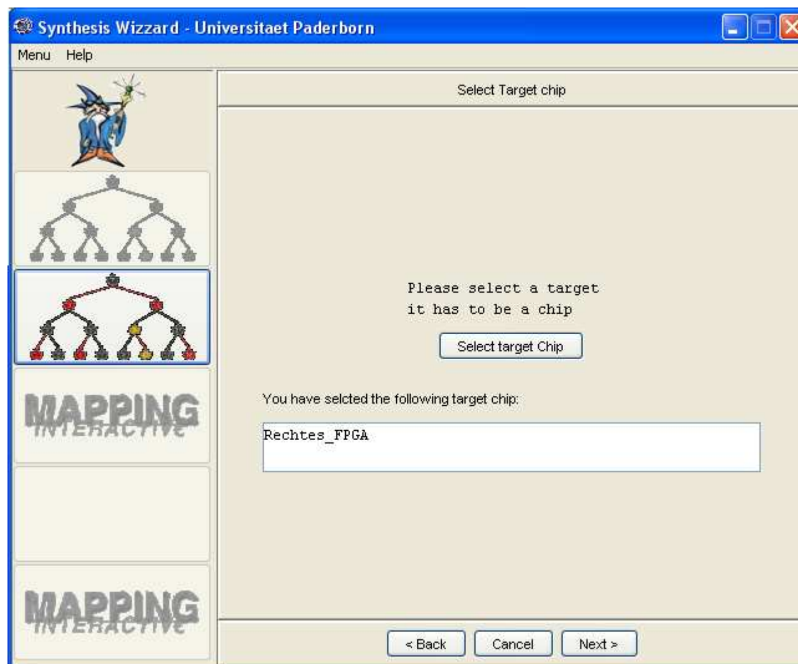

Abbildung A.2.: Synthesis Wizard, Schritt 2 - Auswahl der Zielplattform

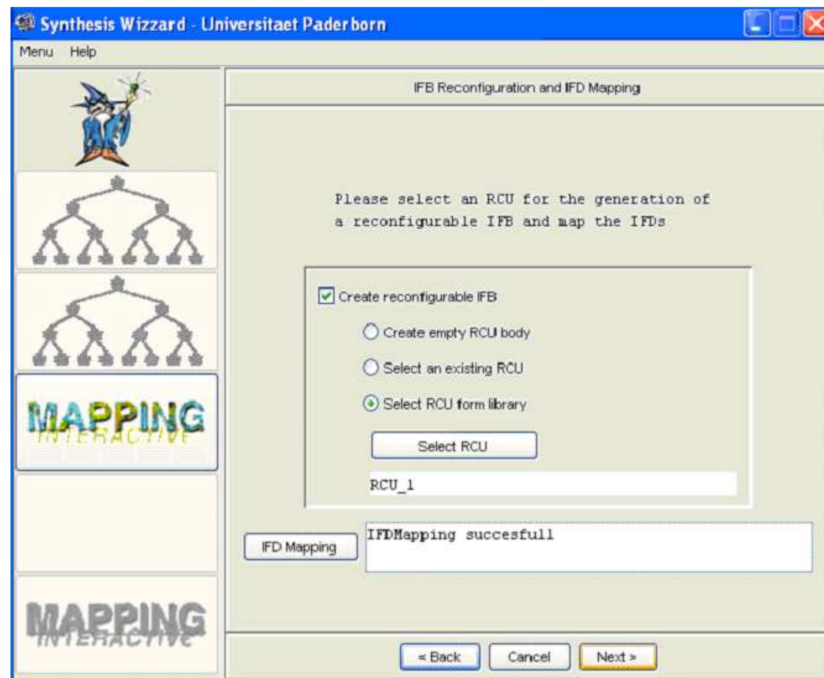

Abbildung A.3.: Synthesis Wizard, Schritt 3 - Definition der Datenabbildung und Festlegung der Rekonfigurierbarkeit des IFBs

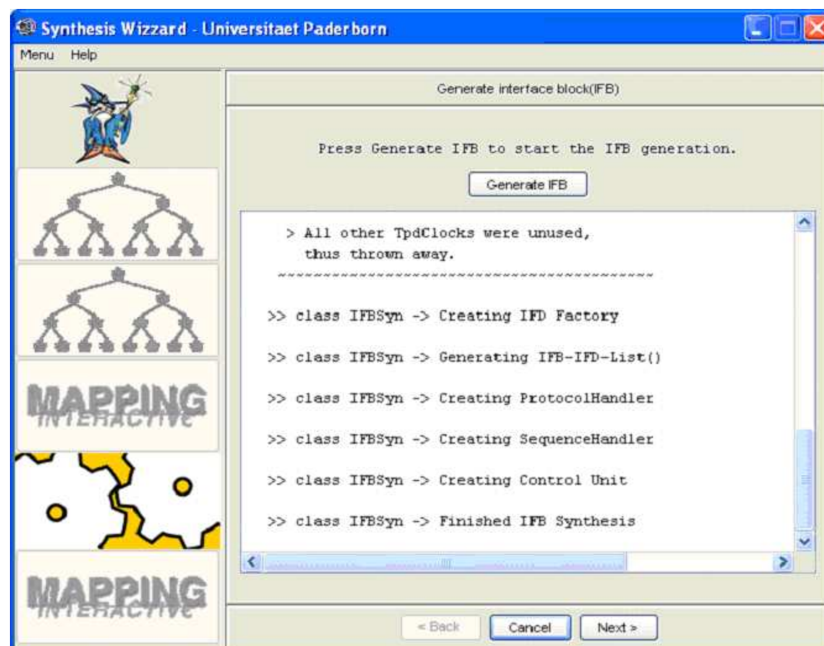

Abbildung A.4.: Synthesis Wizard, Schritt 4 - Erzeugung des IFBs

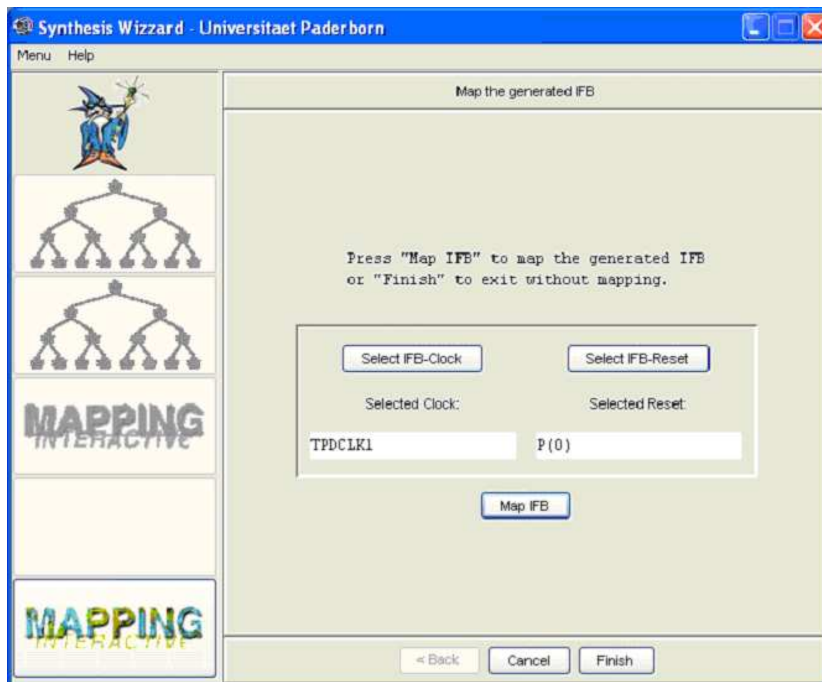

Abbildung A.5.: Synthesis Wizard, Schritt 5 - Einbinden des IFBs

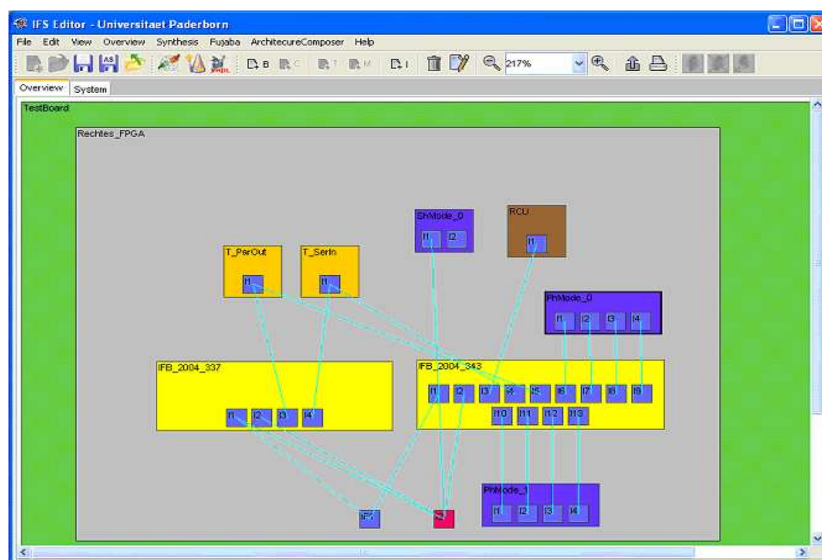

Abbildung A.6.: Der generierte IFB im IFS-Editor

Literaturverzeichnis

[Die05] Dieter Averberg. *Synthese von deadline-konformen Protokollkonvertern für heterogene verteilte Anwendungen.* 2005.

[Dou96] Douglas J. Smith. *HDL Chip design.* Doone Publications, 1996.

[Ihm03] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. Modeling of Configurable HW/SW-Interfaces. In *RSS2003*, pages 51 – 60, 2003.

[Ihm05a] Ihmor, Stefan and Dittmann, Florian. Optimizing Interface Implementation Costs Using Runtime Reconfigurable Systems . In , 2005.

[Ihm05b] Ihmor, Stefan and Loke, Tobias and Hardt, Wolfram. Synthesis of Communication Structures and Protocols in Distributed Embedded Systems . In , 2005.

[Mar02] Markus Voelter. *Program Generation, A Survey of Techniques and Tools.* http://www.voelter.de/data/presentations/ProgramGeneration.zip, 2002.

[Mic05] Michel C. Kouamo S. *Modellbasierte Spezifikation von Schnittstellen im Hardware Entwurf.* 2005.

[OMG03] OMG Specs. *Catalog Of OMG Specifications.* http://www.omg.org/technology/documents/sepc_catalog.htm, 2003.

[Pau04] Paul M. Jörg R. *VHDL Eine Einführung.* Pearson Studium, 2004.

[Tob05] Tobias Loke. *Synthese von Kommunikationsstrukturen in verteilten eingebetteten Systemen.* 2005.

[Wik] Wikipedia. *Die freieEnzyklopädie. Online at.* http://de.wikipedia.org/wiki/Hauptseite.