

Vorwort zur Wissenschaftlichen Schriftenreihe "Eingebettete, selbstorganisierende Systeme"

Diese Schriftenreihe präsentiert wissenschaftliche Arbeiten zum Entwurf und zur Selbstorganisation von eingebetteten Systemen. Im ersten Band dieser Schriftenreihe liegt der Fokus auf der Konzeption von "Rekonfigurierbaren Schnittstellen". Der vorliegende zweite Band stellt eine vollständige Methodik für die Adaptierung von inkompatiblen Schnittstellen in komplexen, verteilten, eingebetteten Systemen vor. Diese Methodik wird als Interface Synthese (IFS) bezeichnet. In den vier Beiträgen werden im Einzelnen die Modellierung, die Analyse und Optimierung und die Codegenerierung als methodische Entwurfsschritte untersucht.

Der erste Beitrag behandelt die Modellierung von komplexen Kommunikationssystemen als ersten Schritt des Interface Synthesis Design Flows. Dabei wird ein intuitives UML2.0 Profil sowie die Modelltransformation in unser Synthesewerkzeug, den IFS-Editor, definiert. Der Interface Synthesis Design Flow bietet somit einen durchgängigen Entwurfsfluss von der abstrakten UML Modellierung hin bis zur ausführbaren Hardware. Die beiden folgenden Arbeiten befassen sich mit der Analyse und der Optimierung der Schnittstellensynthese. Im Beitrag von Christian Behler erfolgt eine detaillierte Analyse des Architekturkonzeptes der Schnittstellen. Dies ermöglicht eine Optimierungsbetrachtung der beiden Parameter Latenz und Flächenbedarf für die Implementierung von Schnittstellenadaptoren. Hierfür werden die Techniken Pipelining und partielle Rekonfigurierung eingesetzt. Mit dem dritten Beitrag wird die globale Optimierung der Kommunikation in verteilten, eingebetteten Systemen aufgegriffen, da diese eine zunehmend zentrale Rolle einnimmt. Ziel der Optimierung ist die Reduktion der auszutauschenden Datenmenge zwischen heterogenen, interagierenden Anwendungen sowie die Minimierung der für die Adaptierung der Kommunikation anfallenden Hardwareressourcen. Eine Besonderheit stellt der Einsatz anwendungsspezifischer Protokolle dar, um den Einsatz teurerer Kommunikationssysteme zu vermeiden. Der abschließende Beitrag widmet sich der Codegenerierung und behandelt den letzten Teil des Interface Synthesis Design Flows. Basierend auf dem Prinzip des "Frame Processing" wird ein VHDL Codegenerator vorgestellt, der insbesondere die Eigenschaft der Rekonfigurierbarkeit der Schnittstelle unterstützt. Dies ist von entscheidender Wichtigkeit für den Austausch von Anwendungen in Echtzeitsystemen zur Laufzeit. Als einziges Verfahren auf diesem Gebiet erlaubt der IFB die Ausführung eines deterministischen Verhaltens während der FPGA Rekonfiguration.

Die vorgestellten Beträge beschreiben die wesentlichen Schritte des automatisierten Schnittstellenentwurfs und motivieren den Leser diese Ergebnisse auf weitere Anwendungsbereiche zu übertragen. Den Autoren der einzelnen Beiträge, Christian Behler, Bertrand Defo, Michel Kuamo und Tobias Loke sowie allen Mitwirkenden bei der Erstellung und Umsetzung dieses zweiten Bandes danken wir für den Einsatz und die Unterstützung.

Prof. Dr. Wolfram Hardt und Dipl.-Inf. Stefan Ihmor
Mai 2006

Inhaltsverzeichnis

1	Modellbasierte Spezifikation von Schnittstellen im Hardware Entwurf	
	Dipl.Inf. Michel Carmel Kouamo Sime	3
2	Datenflussoptimierung in rekonfigurierbarer Hardware durch Pipelining	
	Christian Behler	106
3	Synthese von Kommunikationsstrukturen in verteilten Systemen	
	Tobias Loke	156
4	VHDL Codegenerierung für rekonfigurierbare Schnittstellen in eingebetteten Systemen	
	Gilles Bertrand Gnokam Defo	203

Modellbasierte Spezifikation von Schnittstellen im Hardware Entwurf

Dipl.-Inf. Michel Carmel Kouamo Sime

In immer komplexer werdenden eingebetteten Systemen spielt neben der Entwicklung der eigentlichen Funktionalität die Integration einzelner funktionaler Komponenten und damit die Adaptierung von Schnittstellen eine entscheidende Rolle. Dabei ist ein Ziel, den Grad der Automatisierung zur Erzeugung einzelner Schnittstellenadapter zu optimieren. Ausgangspunkt einer jeden Automatisierung ist eine formelle Beschreibung der Eingabe, in unserem Zusammenhang die Modellierung der Schnittstellen der zu adaptierenden Komponenten.

Der nachfolgende Beitrag befasst sich mit dem Entwurf von Schnittstellen im Rahmen des komponentenbasierten Entwurfs von Hardware und bettet diesen in eine UML-Modellierungssystematik ein. Dazu wird das bestehende Interface Synthese Modell (IFS-Format) in ein für den Designer intuitives UML2.0-Profil umgesetzt. Die Integration des Profils in das bestehende Modellierungskonzept wird auf Metamodell-, Modell- und Instanz-Ebene betrachtet. Dabei werden verschiedene Integrationsvarianten diskutiert. Das UML2.0-Profil beschreibt ein Komponentendiagramm der Systemarchitektur, welches dann durch eine Struktur- und zur Verhaltensbeschreibung verfeinert wird.

Weiterhin wird die softwaretechnische Realisierung der Modelltransformation der UML2.0-Diagramme aus dem UML Case-Tool Fujaba in den IFS-Editor, unser Werkzeug zur Synthese von Schnittstellen, erläutert.

Inhaltsverzeichnis

1. Einführung	7
1.1. Motivation	7
1.2. Problemstellung	7
1.3. Aufgabenstellung	8
1.4. Gliederung der Arbeit	8
2. Stand der Technik	9
2.1. UML2.0	9
2.1.1. Warum eine neue Version?	9
2.1.2. Was ist neu an UML2.0	9
2.1.3. Verwendete Diagrammtypen	10
2.1.4. Erweiterungsmechanismen	17
2.2. Das IFS-Format	18
2.2.1. Einführung in Interface Synthese	18
2.2.2. Systemarchitektur	18
2.2.3. IFS-Editor	21
2.3. Das WerkZeug Fujaba	23
2.3.1. Warum Fujaba?	23
2.3.2. Das Fujaba-Metamodell (RT-Component Diagramms)	24
3. Lösungskonzepte	25
3.1. Erweiterung des bestehenden Modellierungskonzepts	25
3.2. Das Modellierungskonzept in UML2.0	26
3.3. Modelltransformation und Lösungsansätze	27
3.3.1. Modelltransformation	28
3.3.2. Lösungsansätze	29
4. Modellierung	33
4.1. Grundgedanken	33
4.2. Aufbau des Modells	33
4.2.1. Strukturbeschreibung	35
4.2.2. Verhaltensbeschreibung	37
5. Konzept und Implementierung der Modelltransformation	47
5.1. Wahl der Transformationsart und des UML Case-Tools	47
5.2. Transformationskonzept	49

5.3.	Implementierung der Modelltransformation	50
5.3.1.	Werkzeug-Kopplung	51
5.3.2.	Parseralgorithmus	53
6.	Zusammenfassung und Ausblick	65
6.1.	Zusammenfassung	65
6.2.	Ausblick	66
A.	Anhang	67
A.1.	Beschreibung eines Beispielmodells	67
A.1.1.	Die grafische Oberfläche	67
A.1.2.	Komponente und Objektdiagramm	67
A.1.3.	Verhaltensbeschreibung	82
A.1.4.	Zustandsautomaten	85
A.2.	Beispiele zur XML Transformation	89
A.2.1.	SAX-Parser-Beispiel	89
A.2.2.	DOM-Parser-Beispiel	90
A.3.	Java Quellcode	92
A.3.1.	Synch2System()	92
A.3.2.	synch2Protocol()	94
A.3.3.	extractType()	101
	Literaturverzeichnis	103

1. Einführung

1.1. Motivation

Der Entwurf eingebetteter Systeme wird durch immer höhere Anforderungen an die Komplexität der Systeme, die Time-To-Market und viele andere Herausforderungen zunehmend aufwändiger. Einzelne Entwickler können nicht mehr alleine vollständige Produkte erstellen und müssen ihren Teilentwurf an bereits entworfene Komponenten adaptieren. Allerdings kann auch die Integration von IPs (Intellectual Property) sehr mühsam sein, wenn die eingesetzten Komponenten keine aufeinander abgestimmten Schnittstellen vorweisen. Mit dem Ziel, die Wiederverwendung (Reuse) von Komponenten zu erleichtern und die Komplexität in Kommunikationssystemen beherrschbar zu machen, ist das Forschungsthema automatisierte Interface Synthese (IFS) an der Universität Paderborn entstanden [Ihm01, Ihm02].

1.2. Problemstellung

Das IFS-Format (Interface Synthesis Format) ist eine Modellierungssprache, die auf einem XML Schema basiert. Um die Schnittstellensynthese zu automatisieren wurde ein Java Werkzeug (IFS-Editor) implementiert [Fic03], welches das Design von Kommunikationssystemen unter Berücksichtigung der Struktur und der Semantik des IFS-Formats ermöglicht. Obwohl die vollständige Modellierung von Kommunikationssystemen bereits abgedeckt ist, ist das Werkzeug erstens nicht sehr verbreitet und zweitens erfordert der Umgang mit dem IFS-Editor einen nicht unerheblichen Lernaufwand.

Ein wesentlicher Faktor für die Akzeptanz einer Modellierungssprache durch einen Designer besteht darin, eine intuitive Beschreibungsform zu wählen. Die Unified Modeling Language (UML) ist eine, durch die Object Management Group (OMG) weltweit standardisierte Modellierungssprache zur Beschreibung objektorientierter Modelle [Uni04] mit einem hohen Bekanntheitsgrad. UML wird von vielen Werkzeugen mit einer (meist) einheitlichen grafischen Notation unterstützt [Jec04a] und bietet als formale Sprache die Möglichkeit zur Verifikation [Bec03, Hol03]. Die Version UML2.0 verfügt über eine erweiterte Semantik sowie zusätzliche Modellierungselemente und Diagramme [Ber04], die weitere Modellierungsaspekte wie Schnittstellenbeschreibungen und Zeitverhalten ermöglichen [Jec04b]. Die genannten Vorteile legen es nah UML2.0 als Modellierungssprache für die in dieser Arbeit betrachteten Kommunikationssysteme zu nutzen.

1.3. Aufgabenstellung

Die Modellierungssprache der Interface Synthese, das IFS-Format, basierte bis jetzt auf einem XML Schema. Um eine standardisierte Nutzung des IFS-Formats zu ermöglichen, soll in dieser Arbeit das bestehende Modellierungskonzept von XML auf ein zu definierendes UML2.0 Profil übertragen werden. Dabei sollen die Neuheiten von UML2.0 wie z.B. Structured Classes berücksichtigt werden. Im Rahmen des Profils ist zu analysieren, welche Diagrammtypen von UML2.0 als geeignet erscheinen. Basierend darauf soll ein intuitives Modellierungskonzept erarbeitet werden, das in dem Case-Tool Fujaba (From UML to Java and back again [Sve04, Uni]) umgesetzt wird. Anschließend ist die Transformation von UML2.0 in das bestehende IFS-Format zu definieren. Diese soll dann als Werkzeugkopplung zwischen Fujaba und dem IFS-Editor implementiert werden.

1.4. Gliederung der Arbeit

Diese Arbeit ist in sechs Kapitel gegliedert.

Das folgende Kapitel betrachtet die zur Lösung der Aufgabenstellung relevanten Grundlagen. Dabei wird kurz auf das bestehende Interface Synthese Format (IFS-Format) eingegangen. Danach werden die für die Modellierung in UML benötigten Diagrammtypen eingeführt und wichtige Aspekte näher erläutert.

Das dritte Kapitel beschreibt das aktuelle IFS Modellierungskonzept und erläutert die Methodik für eine intuitive Modellierung in UML2.0. Danach werden mögliche Konzepte und Realisierungen zur Modelltransformation von UML2.0 in das bereits bestehende XML Format vorgestellt.

Im Anschluss werden in Kapitel vier die relevanten Aspekte des UML2.0 Profils zur Modellierung des IFS-Formats beschrieben. Ein ausführliches Beispiel dazu findet sich im Anhang.

Kapitel fünf stellt die Implementierung der Modelltransformation in das XML basierte IFS-Format ausgehend von dem internen Hauptspeichermodell des Werkzeuges Fujaba vor.

Kapitel sechs fasst die Ergebnisse zusammen und gibt einen Ausblick.

Im Anhang wird ein detailliertes Beispiel einer UML2.0 Modellierung geliefert. Hier werden die erlaubten Parameter und Abkürzungen innerhalb des Modells angegeben.

2. Stand der Technik

Dieses Kapitel vermittelt einen Einblick in den aktuellen Stand der Technik. Gleichzeitig dient es als Grundlage zum Verständnis der späteren Kapitel.

2.1. UML2.0

Die Unified Modeling Language (UML) ist eine durch die Object Management Group (OMG) weltweit standardisierte Modellierungssprache zur Beschreibung objektorientierter Modelle zur Spezifikation, Visualisierung, Konstruktion und Dokumentation komplexer Softwaresysteme. Durch den Einsatz dieser Standardmodellierungssprache ergibt sich die Möglichkeit für die Entwickler, ihren Entwurf oder die Entwicklung von Softwaremodellen auf einheitlicher Basis zu diskutieren.

2.1.1. Warum eine neue Version?

In der Version UML1.5 war die Modellierung bestimmter Aspekte in eingebetteten Systemen sehr aufwendig. Der Markt hat sich mit der Evolution entwickelt und neue Programmiersprachen sowie neue Anwendungsbereiche sind entstanden. UML1.5 bot für manche Einsatzgebiete zu wenig Konstrukte, die dann durch komplexe Konstrukte ausgedrückt werden mussten. Als weltweite Standardmodellierungssprache hat sich mit der Zeit und durch die Erfahrungen der Anwender mit UML, die Notwendigkeit der Veränderung und Erweiterung des UML Metamodells ergeben. Unter anderem wurden Sprachen, die in ihrer Domäne wiederum den Standard darstellten, wie z.B. SDL, ein Teil der UML2.0.

2.1.2. Was ist neu an UML2.0

Zusätzlich zu den marginalen Änderungen bei den Klassendiagrammen, Objektdiagrammen und Use-Case-Diagrammen, sowie kleinen Änderungen bei den Paketdiagrammen und Verteilungsdiagrammen, ebenso wie die bedeutenden Änderungen bei den Sequenzdiagrammen, Zustandsautomaten und Aktivitätsdiagrammen sind die folgenden Diagramme neu eingefügt worden.

2. Stand der Technik

- Interaktionsübersichtsdiagramm
- Kompositionsstrukturdiagramm
- Timing-Diagramm
- Kommunikationsdiagramme

Interaktionsübersichtsdiagramm

Durch die Verknüpfung von Sequenz- und Aktivitäts-Diagrammen lassen sich unterschiedliche Verhaltensdiagramme auf Top-Levelebene übersichtlich darstellen [Pat03].

Kompositionsstrukturdiagramm

Das Kompositionsstrukturdiagramm erlaubt es, die innere Struktur verschiedener UML-Elemente wie Klassen, Use-Cases und Aktivitäten einheitlich darzustellen. Eine präzise Modellierung der Teilbeziehungen über spezielle Schnittstellen (Ports) ist ebenso möglich. Zusätzlich können Aspekte der Zusammenarbeit der verschiedenen Modellelemente berücksichtigt werden [Jec04b].

Timing-Diagramm

Die neu in UML2.0 eingefügten Timing-Diagramme ermöglichen eine präzise Beschreibung des Zeitverhaltens von Objekten und Systemen. Diese sind geeignet für die Detailbetrachtungen, bei denen es wichtig ist, dass ein Ereignis zum richtigen Zeitpunkt eintritt [Jec04b].

Kommunikationsdiagramme

Das ehemalige Kollaborationsdiagramm ist eine Teilmenge des Sequenzdiagramms, das den kooperativen und weniger den zeitlichen Aspekt des Nachrichtenaustauschs zwischen Kommunikationspartnern veranschaulicht [Jec04b].

2.1.3. Verwendete Diagrammtypen

In dieser Arbeit wird die UML2.0 benutzt um Kommunikationssysteme zu modellieren. Das daraus resultierende Modell dient als Input für eine automatisierte Synthese. Wie in Abbildung 2.1 dargestellt ist, stehen 13 Diagrammtypen für die Modellierung zur Verfügung, von denen sechs zu der Modellierung der Systemstruktur und die anderen sieben zu der Modellierung des Systemverhaltens dienen. Aus den unterschiedlichen Diagrammtypen wurden vier für diese Aufgabe ausgewählt. Diese sind in Abbildung 2.1 in blau gekennzeichnet.

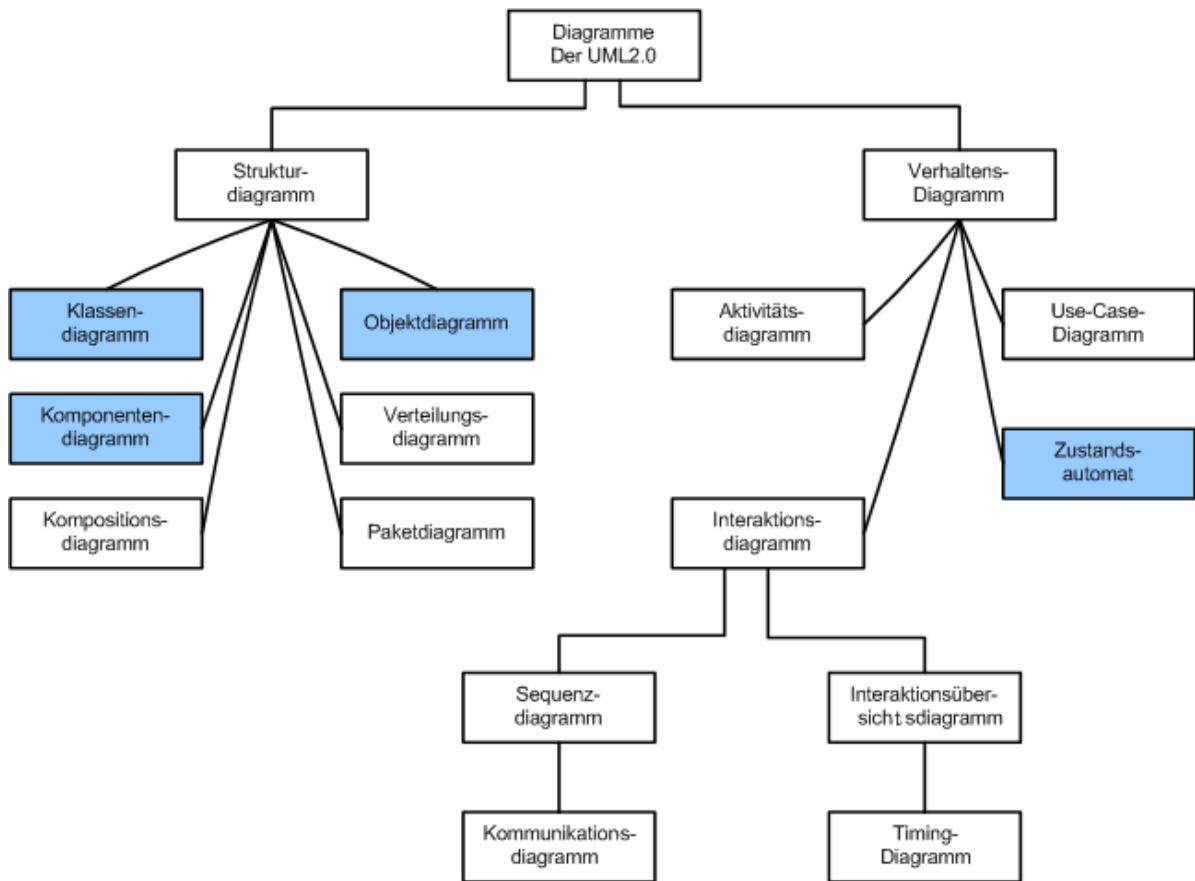


Abbildung 2.1.: Diagrammtypen der UML2.0 [Jec04b]

- **Klassendiagramm**

Klassendiagramme werden verwendet, um statische Eigenschaften eines Systems zu modellieren. Sie enthalten alle relevanten Strukturzusammenhänge, Datentypen (durch Variablen) und bilden durch die Berücksichtigung der Operationen die Brücke zu den dynamischen Diagrammen [Pat04].

- **Objektdiagramm**

Das Objektdiagramm enthält genau eine konkrete Ausprägung (Objekt und deren Attributbelegung) eines Klassendiagramms [Tho04].

- **Komponentendiagramm**

Das Komponentendiagramm illustriert die Organisations- und Abhängigkeitsstruktur einzelner technischer Systemkomponenten [And04]. Das Komponentendiagramm erbt von dem Klassendiagramm. Die Abbildung 2.2 stellt diese Vererbungsbeziehung zwischen *Class*- und *Component*-Klassen vor, wie sie im UML-Meta-Modell definiert ist. Die *Component*-Klasse hat eine Kompositionsbeziehung zu der *Realization*-Klasse die wiederum eine Assoziationsbeziehung zu der *Classifier*

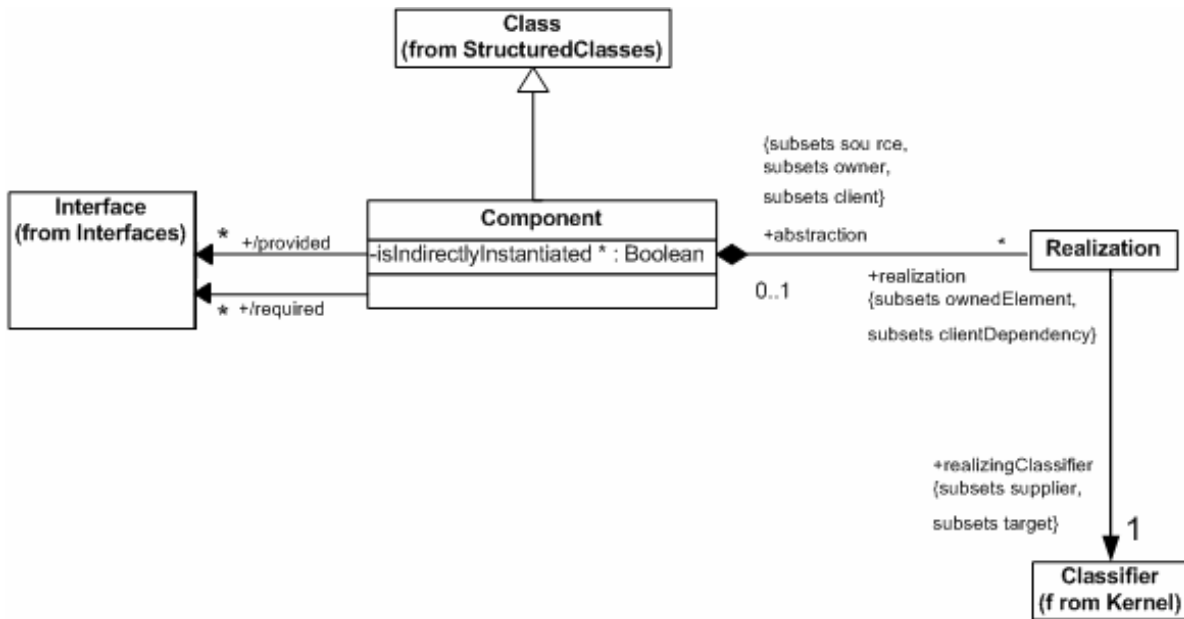


Abbildung 2.2.: Grundlegende Basis der Komponentenstruktur [Uni04]

Klasse hat. Die *Classifier* Klasse ist eine Abstraktion des UML objektorientierten Grundkonzeptes “Klasse” auf einer weiteren Ebene, die innerhalb des Sprachdesigns der UML2.0 eine zentrale Rolle spielt. Innerhalb der UML-Spezifikation wird immer wieder auf diesen Begriff Bezug genommen, obwohl er nicht als direkt nutzbares grafisches Konstrukt für den Modellierer zur Verfügung steht.

Die Schnittstellen für die Kommunikation zwischen den Komponenten lassen sich durch die Begriffe *provided* und *required interfaces* ausdrücken und werden mit einer Interfaceklasse versehen. Die Klasse Interface wird hierbei entweder als *provided interfaces* gekennzeichnet, wenn sie durch die Komponente angeboten wird, oder als *required interfaces*, wenn sie von der Komponente benötigt wird (vgl. Abbildung 2.2).

Für die nachfolgende Arbeit spielt das Komponentendiagramm bei der Modellierung von Kommunikationssystemen eine wichtige Rolle. Es beschreibt die oberste Ebene des in dieser Arbeit entwickelten UML-Modells, von dem aus alle weiteren Diagramme (Klassendiagramme und Protokollzustandsautomaten) referenziert werden. Im Folgenden werden Aspekte und Begriffe erläutert, die bei der Modellierung von Komponentendiagrammen verwendet werden.

Schnittstellen

Grundsätzlich gibt es zwei Arten von Schnittstellen. Diese sind in Abbildung 2.3 veranschaulicht. Komponenten werden je nach Aufgabe direkt, oder indirekt über einen Port, verknüpft (siehe Tabelle 2.1). Die direkte Variante, welche ohne Ports auskommt,

wurde bereits im Unterpunkt “Komponentendiagramm” erwähnt (vgl. Abbildung 2.2). Die indirekte Variante wird bei der Portbeschreibung vorgestellt.

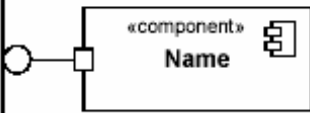
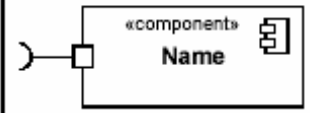
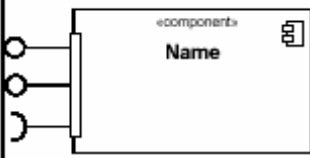
<i>NODE TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Component has provided Port (typed by Interface)		See “Port”.
Component uses Interface		See “Interface”.
Component has required Port (typed by Interface)		See “Port”.
Component has complex Port (typed by provided and required Interfaces)		See “Port”.

Tabelle 2.1.: Verfügbare UML Schnittstellen [Uni04]

- Implementierte Schnittstelle (*provided interface*)

Das Symbol, wie es in Tabelle 2.1 in der ersten Zeile dargestellt ist, definiert, dass eine gegebene Schnittstelle implementiert wird.

- Benötigte Schnittstelle (*required interface*)

Das Symbol in der zweiten und dritten Zeile von Tabelle 2.1 definiert, dass Schnittstellen benötigt werden. Die Tabelle 2.1 gibt einen Überblick über die mögliche Einbindung von Schnittstellen in Komponenten.

<i>PATH TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Assembly connector		See “assembly connector”. Also used as notation option for wiring between interfaces using Dependencies.

Tabelle 2.2.: Portbeschreibung in UML2.0 [Uni04]

Schnittstellen stehen immer in Relation mit dem *interface*. Dort werden Attribute und Operationen deklariert [Jec04b]. Die Abbildung 2.3 zeigt ein Beispiel eines Komponentendiagramms. Die Komponente “*Komponente1*” bietet nach außen eine implementierte Schnittstelle für bestimmte Aufgaben an. Gleichzeitig benötigt diese Komponente zur Erfüllung ihrer Aufgaben zwei Schnittstellen, von denen eine Schnittstelle durch die Komponente “*Komponente2*” über einen Port zur Verfügung gestellt wird. Die benötigte und implementierte Schnittstelle wird über einen *Konnektor* verbunden (siehe Tabelle 2.2).

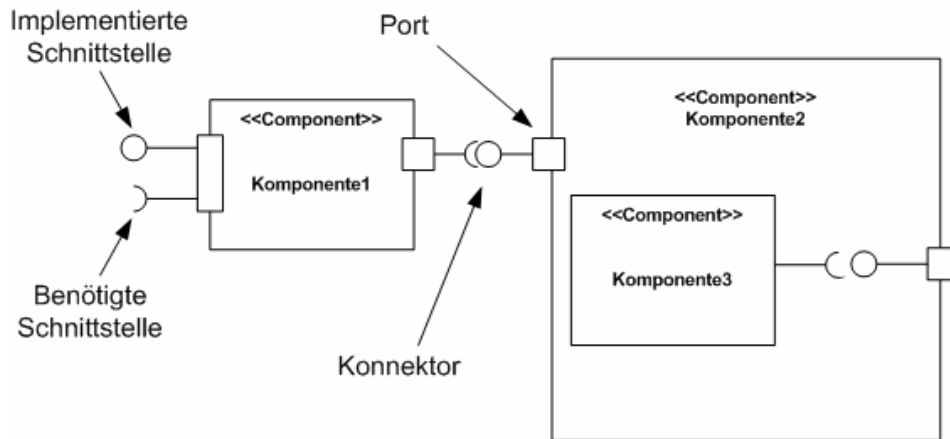


Abbildung 2.3.: Notation eines Ports mit Schnittstelle [Jec04b]

Port

Ein Port wird als kleines Quadrat symbolisiert und spezifiziert die Stelle, an der eine Interaktion zwischen Komponenten stattfindet. Jeder Port wird durch eine Klasse repräsentiert. Die Klasse hat eine Kompositionsbeziehung zur dem *EncapsulatedClassifier* und hat wiederum abgeleitete Interfaces. Das sind die *required* und *provided Interfaces* (vgl. Abbildung 2.4). Die Schnittstellen für die indirekte Kommunikation werden hierbei über Ports realisiert. Die möglichen Darstellungen werden in Tabelle 2.1, Zeile 1,3 und 4 dargestellt. Alle Teile und Einheiten des umschließenden *Classifiers*, die zur Kommunikation fähig sind, werden in der *ConnectableElement* Klasse verwaltet und von der Portklasse geerbt.

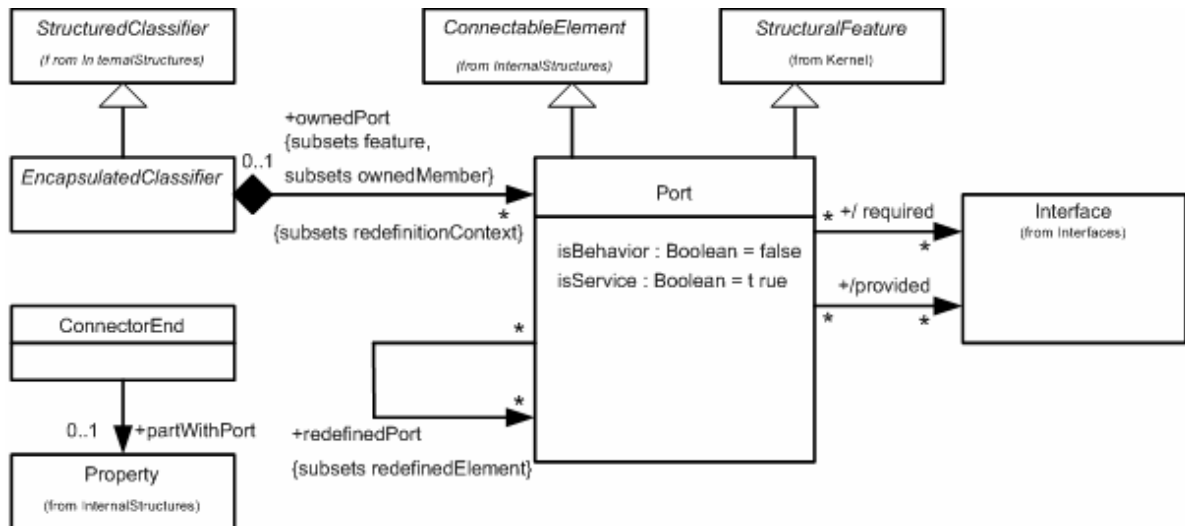
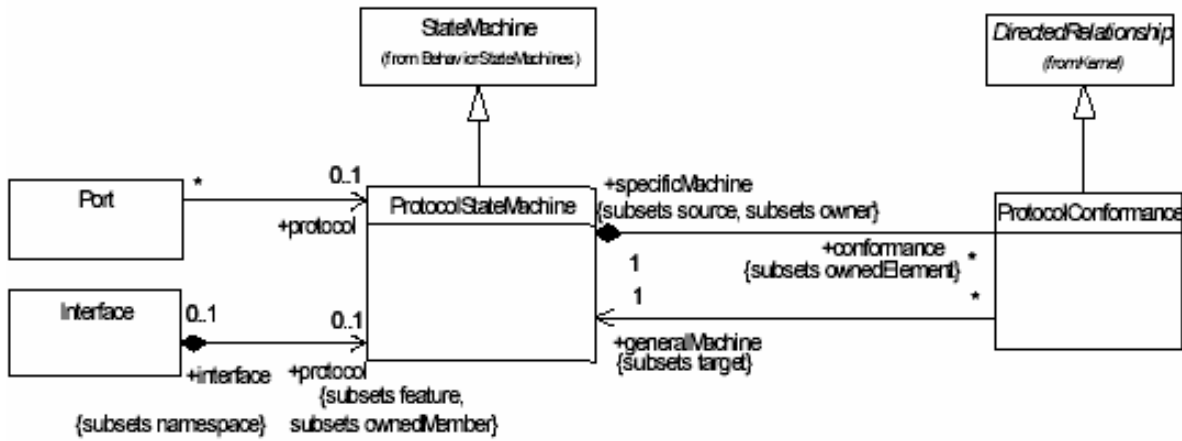


Abbildung 2.4.: Port Metamodell [Uni04]

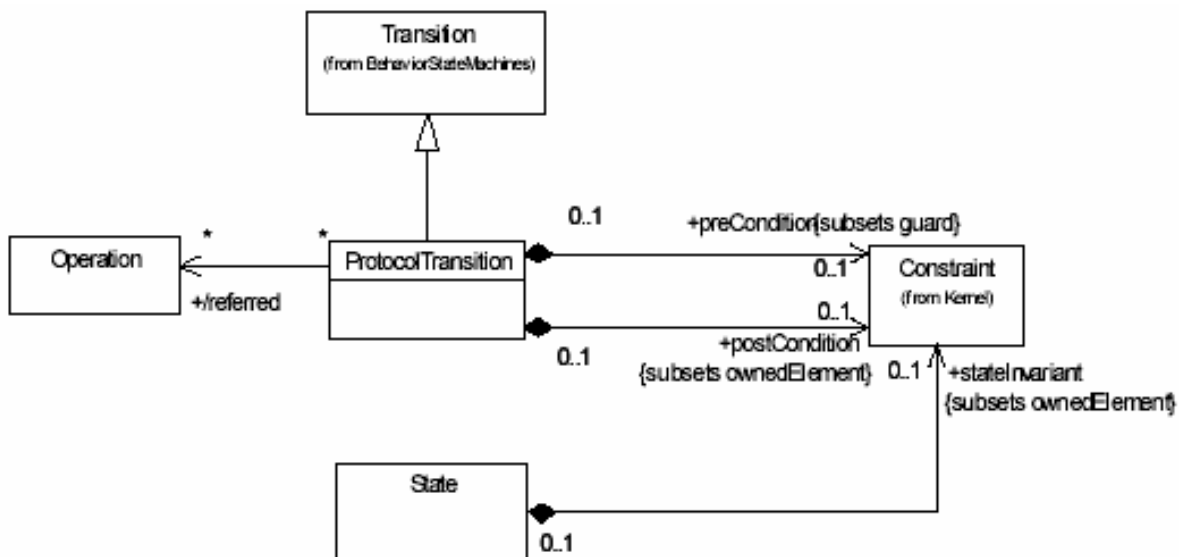
- Zustandsautomat

Das UML-Metamodell der *ProcolStateMachine* wird in Abbildung 2.5 dargestellt. Die *ProcolStateMachine* (Protokoll Zustandsmaschine) legt fest, welche Operationen eines *Classifiers* abhängig vom internen Zustand dieses *Classifiers* aufgerufen werden dürfen. Die *ProcolStateMachine* steht durch die Vererbungsbeziehung in Relation zu der Klasse *StateMachine* (Zustandsautomat). Die Klassen Port und Interface wurden bereits eingeführt. Hier besteht wiederum eine Kompositionsbeziehung zwischen der Interface- und ProcolStateMachine-Klasse und eine Assoziationsbeziehung zwischen Port- und *ProcolStateMachine*-Klasse. Wie bereits bei der Schnittstellenbeschreibung erwähnt (je nach dem, ob eine direkte oder indirekte Kommunikationsschnittstelle zwischen Komponenten besteht), kann das Verhalten der Kommunikation mit der entsprechenden *ProcolStateMachine* realisiert werden (vgl. Abbildung 2.5).



Abbildungung 2.5.: Protokoll Zustandsmaschine [Uni04]

Die Abbildung 2.6 stellt die *ProtocolTransition*-Klasse dar. In der *ProtocolTransition*-Klasse sind die erlaubten Abfolgen von Aufrufen der Operationen definiert, die von einem *Classifier* angeboten werden. Sie steht über eine Assoziationsbeziehung mit der Operationsklasse in Beziehung und erbt von der Transition-Klasse. Die *ProtocolTransition* hat zwei unterschiedliche Kompositionsbeziehungen: die *preCondition* und die *postCondition* zu der *Constraint*-Klasse. Die Constraint-Klasse legt die Bedingung fest, wann ein Transition schalten soll und steht in Kompositions-Beziehung zu der Zustandklasse.



Abbildungung 2.6.: Constraint Klasse [Uni04]

2.1.4. Erweiterungsmechanismen

Es gibt drei Erweiterungskonzepte in der Standardspezifikation, die es ermöglichen UML flexibel zu gestalten. Die Erweiterungsmöglichkeiten sind Stereotypen, Tagged Values und Profiles. Stereotypen und Tagged Values dienen zu einer allgemeinen Erweiterung von Ausdrucksmöglichkeiten. Sie bieten dem Benutzer die Möglichkeit neue Terminologien zur Sprache hinzuzufügen. Eine Änderung des UML-Metamodells erfordert ein sehr gutes Grundverständnis des UML-Metamodells. Kleine Änderungen haben große Auswirkungen und es gibt nur wenige Tools, die die Veränderung des UML-Metamodells erlauben.

- **Stereotypen**

Der Sprachvorrat der UML lässt sich mit Hilfe von Stereotypen je nach Bedürfnis eines konkreten Projektes erweitern bzw. anpassen. Als Metamodellklasse beschreiben Stereotypen, wie andere Metamodellklassen erweitert werden. Sie werden in spitze Klammern gesetzt (vgl. Abbildung 2.7) [Jec04b]

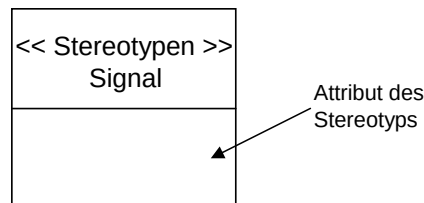


Abbildung 2.7.: Definition des Stereotyps

- **Tagged Values**

Ein Tagged Value definiert gewisse Eigenschaften von UML Elementen und besteht aus Name (Tag) und dem Wert (Value). Er wird in geschweiften Klammern abgebildet [Jec04b].

- **Profiles**

Ein Erweiterungsmechanismus für UML wurde mit dem Konzept der Profile geschaffen, das die Anpassung und Wiederverwendung der UML auf bestimmten Anwendungsgebieten ermöglicht. Profile sind Pakete, die Konstrukte zur Erweiterung von Metamodellen enthalten. Sie fügen keine neue Semantik in die Standardspezifikation des UML ein und müssen innerhalb der UML Semantik bleiben [Uni04]. Profile definieren auf einer höheren Ebene neue Aspekte. In einem Profil werden Regeln definiert, wie bestimmte UML Elemente in bestimmten Kontexten zu interpretieren sind. Die Standardspezifikation von UML wird dadurch verfeinert, ohne eine neue Semantik innerhalb der UML Semantik einzufügen [Boo].

2.2. Das IFS-Format

Das IFS-Format (Interface Synthesis Format) ist eine Modellierungssprache, die auf einem XML Schema [W3C02] basiert. Die IFS ermöglicht eine automatisierte Synthese eines Interface Blocks (IFB). Das Konzept des IFB wurde erdacht, um Schnittstellen zwischen unterschiedlichen Kommunikationskomponenten automatisiert generieren zu können, ohne Änderungen an den Komponenten selbst vorzunehmen [Ihm01]. Das IFS-Format ist in hierarchische Module aufgeteilt und ermöglicht die Wiederverwendung von einzelnen Modulen an verschiedenen Stellen.

2.2.1. Einführung in Interface Synthese

Das IFS-Format ermöglicht es komplexe Kommunikationssysteme unter Berücksichtigung der physikalischen Struktur mit ihren elektronischen Eigenschaften sowie der logischen Protokollsicht zu modellieren. Dazu gehören Systemarchitektur-, Schnittstellen- und Zielplattformbeschreibungen. Die Informationen werden benötigt, um eine automatisierte Synthese eines IFB zu ermöglichen. Die beschriebenen Kommunikationssysteme bestehen aus verschiedenen Boards, Chips, Tasks und Medien, die über Schnittstellen miteinander kommunizieren. Die Funktionalität des Systems wird innerhalb der Tasks und Medien realisiert. Medien, z.B. Busse wie USB oder PCI, können auf allen Hierarchieebenen vorkommen und sind nicht mit einer einfachen Verdrahtung zu verwechseln [Fic03].

Die Schnittstellen- und Zielplattformbeschreibungen sowie die Abbildung der auszutauschenden Daten (*IFD-Mapping*) werden für die Synthese benötigt. Das *IFD-Mapping* wird allerdings erst für die Synthese benötigt.

Da im Rahmen dieser Diplomarbeit die Modellierung im Vordergrund steht, wird zunächst der aktuelle Stand der Technik bezüglich des Modellierungsaspektes näher erläutert.

2.2.2. Systemarchitektur

Die Systemarchitektur ist ein komplexes Kommunikationssystem, die alle Systemkomponenten beinhaltet, die zur Interface-Synthese benötigt werden. Das System selbst steht auf der obersten Ebene der Systemarchitektur, gefolgt durch Board, Chip, Task und Medium. Dazu kommt noch der IFB, das Ergebnis der Schnittstellensynthese. Die Systemarchitektur lässt sich in zwei Gruppen teilen. Die Architekturkomponenten zur Beschreibung der Architektur und die Kommunikationskomponenten zur Beschreibung der Kommunikation. Jede Komponente hat eine Identifikation, die eine eindeutige Identifizierung einzelner Komponenten ermöglicht. Die Version beinhaltet die Versionsinformation. Die Target Platform Description (TPD) ermöglicht die Beschreibung von Chips mit ihren nutzbaren Ressourcen. Dazu zählen unter anderem die Clock-Netzwerke, die bei der Implementierung der Interface Blöcke (IFB) genutzt werden können. Die Kommunikationskomponenten sowie die Architekturkomponenten verfügen über eine Liste

von Schnittstellen (Interfaces), die für die Kommunikation zwischen den Systemkomponenten zur Verfügung stehen. Die Verdrahtung der Schnittstellen untereinander erfolgt über die InterfaceMaps. Dabei kann eine Systemkomponente mit einer anderen Systemkomponente verbunden werden. Folgende Abbildung zeigt eine gesamte Aufteilung der Systemarchitektur. Eine ausführliche Beschreibung des IFS-Formats wurde in der Studienarbeit von Oliver Fick [Fic03] vorgestellt.

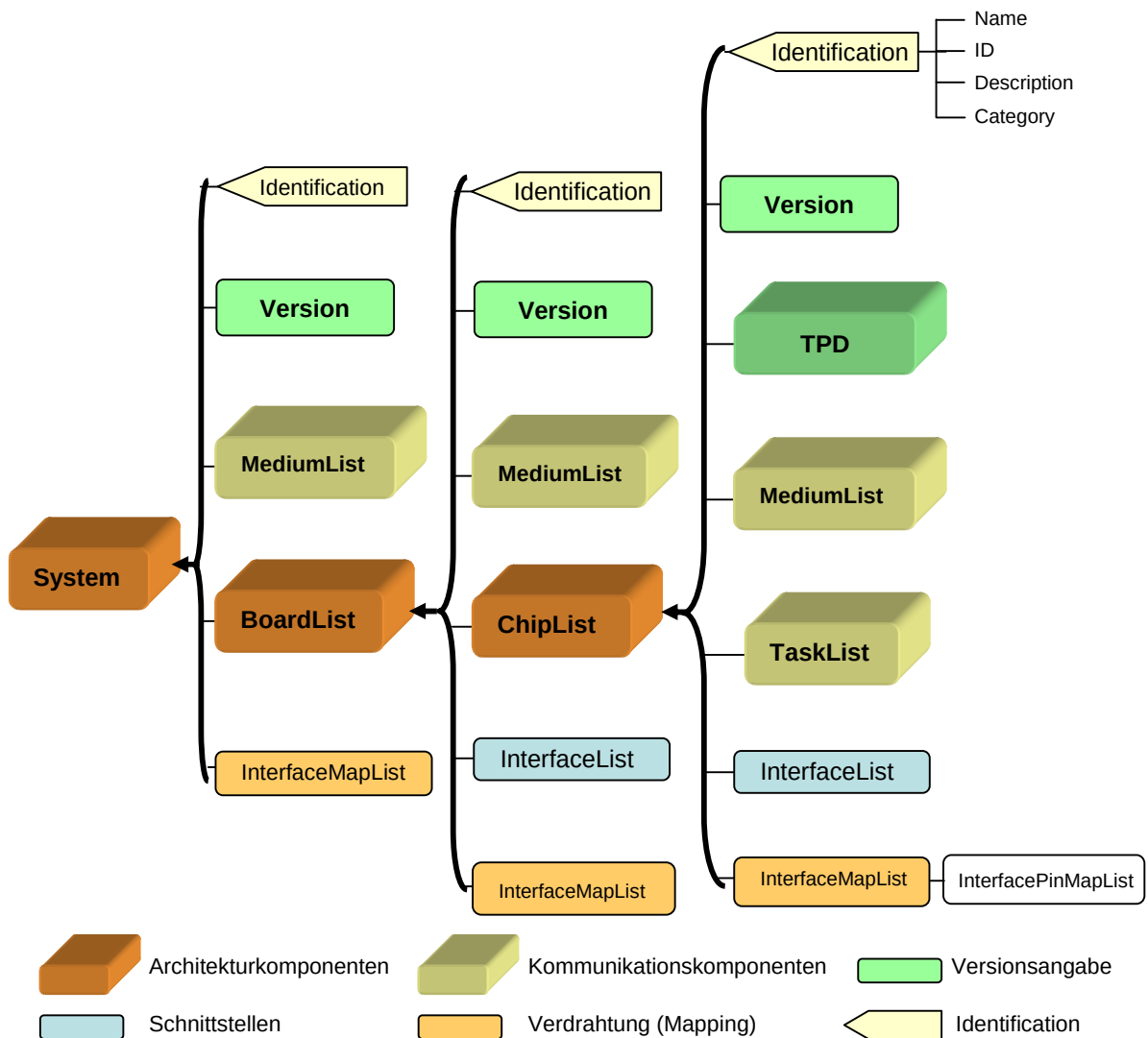


Abbildung 2.8.: Architektur- und Kommunikationskomponenten

Architekturkomponenten

- **System**

Das System stellt die oberste Ebene des IFS-Formats dar und repräsentiert das gesamte Kommunikationssystem. Als Listen von Unterkomponenten beinhaltet das System die BoardList, MediumList und InterfaceMapList. Wie bereits erwähnt, haben alle Komponenten der Systemarchitektur Parameter, die ein eindeutiges Identifizieren und Referenzieren ermöglichen. Diese Parameter werden in der Identification zusammengefasst.

- **Board**

Boards repräsentieren Plattformen wie FPGA-Boards, auf denen sich elektrotechnische Bausteine, wie Chips, befinden. Das IFS-Format ermöglicht die Beschreibung eines Boards durch die Komponenten MediumList, ChipList, InterfaceList und InterfaceMapList, zusätzlich zu dem Parameter Identification.

- **Chip**

Chips repräsentieren Implementierungsplattformen, die unterschiedliche Aufgaben innerhalb eines Boards übernehmen können. Die Tasks, sowie Medien (OnChip-Bus) und IFBs werden in Chips implementiert. Ein Chip wird durch eine Menge von Parametergruppen und Sub-Schemata zusammengesetzt: Neben der Identification und der Zielplattformbeschreibung TPD beinhalten Chips Listen von Medien, Tasks, Interfaces und InterfaceMaps.

Kommunikationskomponenten

- **Task**

Die unterste Hierarchiestufe der Systemarchitektur sind die Tasks. Als Ergebnis der Synthese werden auch die IFBs hier abgelegt (siehe Abbildung 2.9). In Tasks werden Algorithmen für die jeweils einzelne Funktionalität des Gesamtsystems implementiert. Ebenso wie das System, Board und Chip beinhalten die Tasks die Identifikationsattribute und Schnittstellen. Dazu kommt noch eine Liste von Protokolls und der Protokollmaps. Mit Hilfe der Map werden einzelne physikalische Pins der Schnittstelle logischen Protokollpins innerhalb der Protokollbeschreibung zugeordnet.

- **Medium**

Wie man in Abbildung 2.8 sieht, können Medien auf allen Hierarchieebenen vorkommen. Ein Medium ist eine komplexe Systemkomponente, wie eine Task mit physikalischen Eigenschaften, Struktur (Geometrie) der Schnittstelle und einem Protokoll.

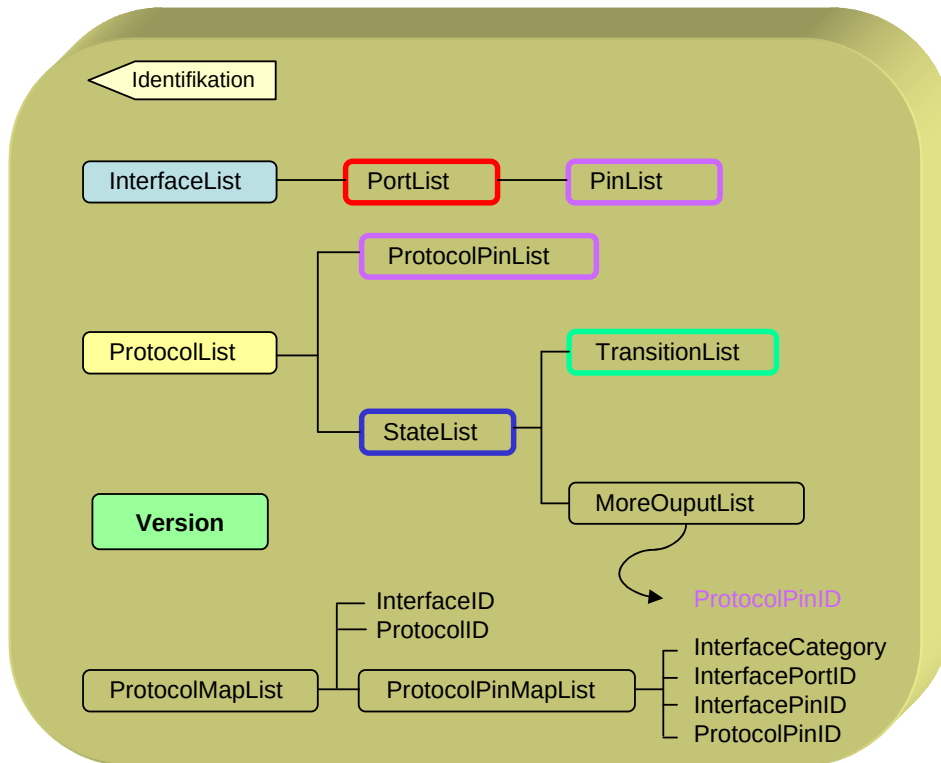


Abbildung 2.9.: Task im IFS-Konzept

2.2.3. IFS-Editor

Der IFS-Editor ist ein Java Programm (IFS-Editor), das ein modellbasiertes Design des IFS-Schemas ermöglicht. Dabei werden die Struktur und die logischen Zusammenhänge des IFS-Formats berücksichtigt. Die Übertragung des bestehenden XML Schemas auf Java wurde mit Hilfe einer objektorientierten Datenstruktur realisiert. Dadurch wird das Laden und Speichern erstellter Instanzen der Systemarchitektur durch den IFS-Editor leicht möglich. Die grafische Oberfläche (*GUI, graphical user interface*) des IFS-Editors repräsentiert den Aufbau des IFS-Formats und teilt sich in mehrere getrennten Sichten auf. Jede dieser Sichten repräsentiert ein bestimmtes Teil-Schema des IFS-Formats. Dort werden die Parameter des zugehörigen XML Schemas editiert.

Die grafische Oberfläche des IFS-Editors (vgl. Abbildung 2.10) illustriert beispielhaft eine Systemarchitektur, die aus einem *System* "sys1" besteht. Diese Abbildung repräsentiert die Systemsicht mit zwei *Boards* b1 und b2, zwei *InterfaceMaps* für die Verdrahtung der beiden *Boards* und die Beschreibung (*Description*) des bestehenden *Systems*.

Auf der Abbildung 2.11 kann man weitere Teilkomponenten des *Systems sys1* erkennen. Diese Oberfläche stellt unter anderem die Sicht zu dem *Board* b1 dar. Das *Board* b1 beinhaltet einen *Chip*, ein *Medium*, zwei *Interfaces*, drei *InterfaceMaps* und eine *Description*.

2. Stand der Technik

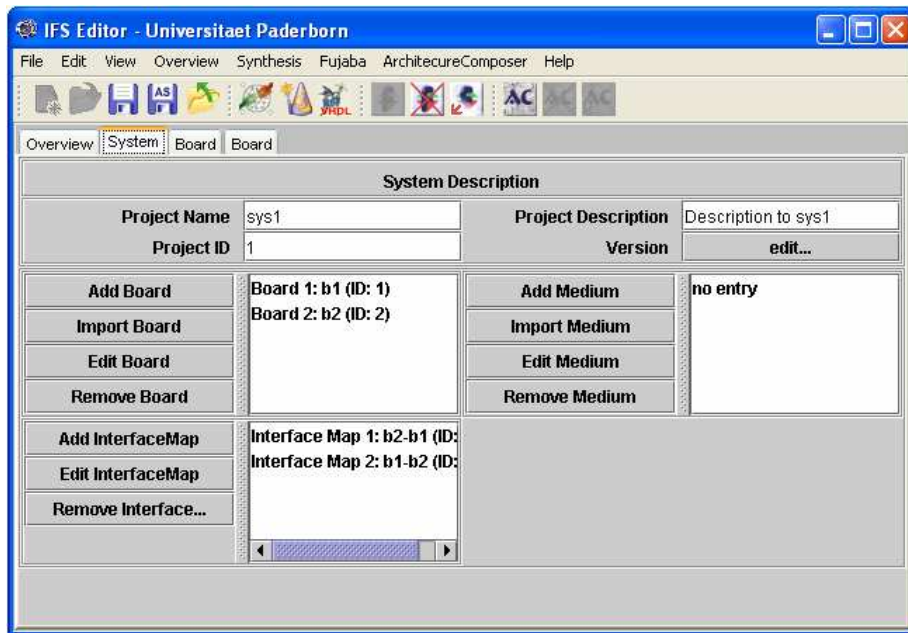


Abbildung 2.10.: IFS-Editor

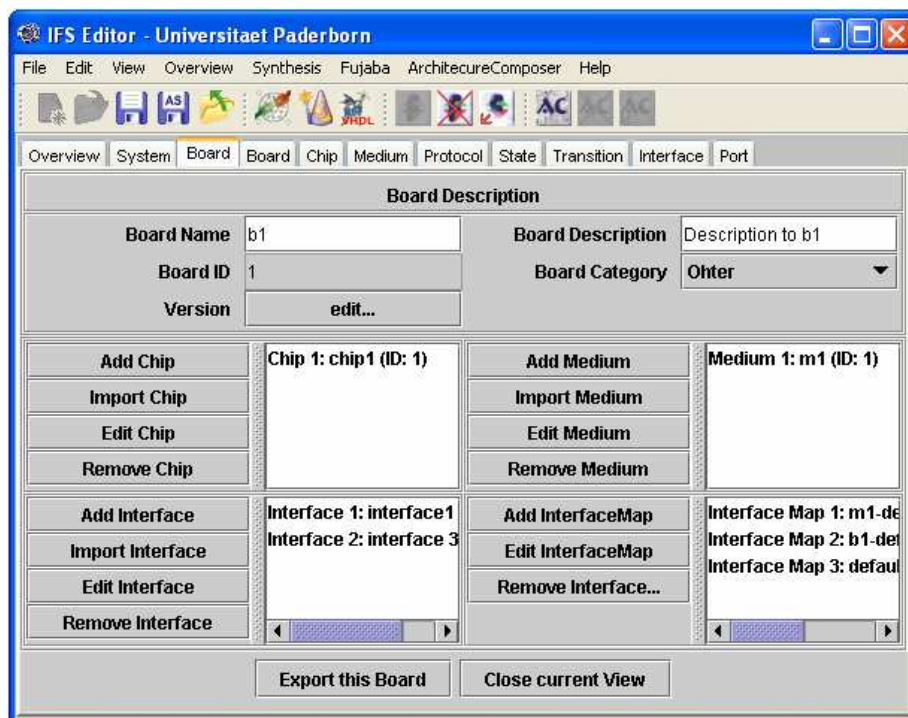


Abbildung 2.11.: IFS-Editor Board-Sicht

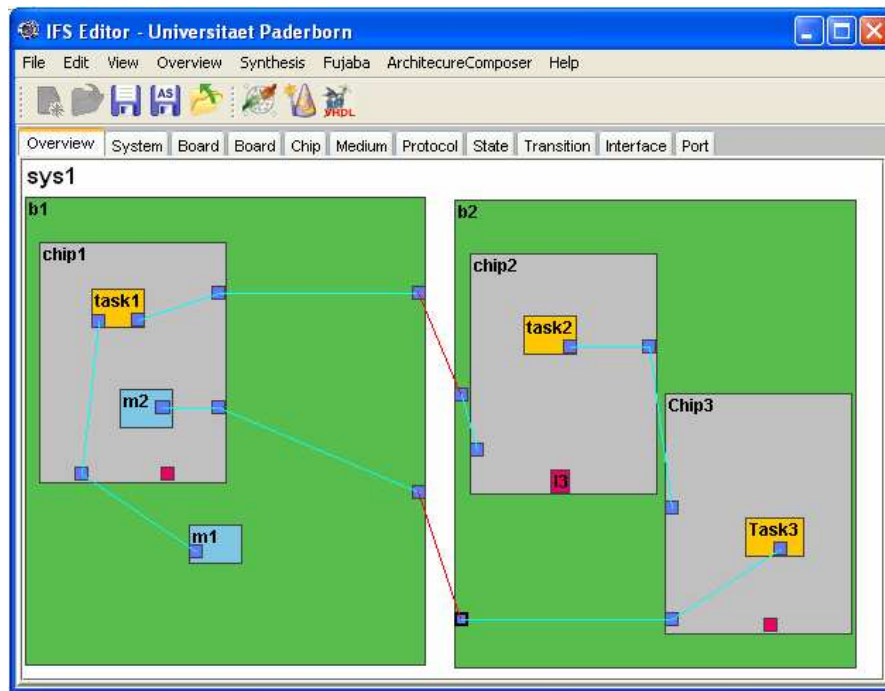


Abbildung 2.12.: IFS-Editor Overview

Der *Overview* des IFS-Editors, wie er in Abbildung 2.12 dargestellt ist, visualisiert das gesamte System. Die grafische Ansicht im IFS-Editors ist ähnlich zu der grafischen Notation der UML Komponentendiagramme.

2.3. Das WerkZeug Fujaba

Fujaba (From UML to Java and back again) wurde in der AG Softwaretechnik an der [Uni] Universität Paderborn entwickelt und ist eine Entwicklungsumgebung, die es ermöglicht aus UML-Spezifikationen Java-Code zu generieren. Weiterhin ist es möglich, aus bestehendem Quellcode die entsprechenden UML-Spezifikationen wieder zurück zu gewinnen (Reengineering).

2.3.1. Warum Fujaba?

Mit der Standardisierung der UML (Unified Modeling Language) durch die OMG (Object Management Group) ist eine einheitliche Notation für die objekt-orientierte Modellierung entstanden. Für die objektorientierte Modellierung existiert eine Menge von UML-Tools wie Rational Rose [Rat, Rat99], Together J [Bor] oder Fujaba [Uni] die unterschiedliche Stärken und Schwächen haben [Jec04a]. Der Fujaba-Editor ist ein Entwurf der Universität Paderborn. Der Programmcode liegt für Forschungszwecke vor. Es beinhaltet alle UML2.0 Komponenten, die für diese Arbeit wichtig sind. Der direkte Zugriff auf die interne Datenstruktur des ausführbaren Programmcodes ist möglich.

2.3.2. Das Fujaba-Metamodell (RT-Component Diagramms)

Die Entwicklungsumgebung FUJABA bietet die Möglichkeit Systeme mit Hilfe von unterschiedlichen Diagrammen und Diagrammelementen zu modellieren. Für die einzelnen Diagrammart wurde ein werkzeugspezifisches Metamodell zugrunde gelegt. Alle erstellten Diagramme werden durch Instanzen der Metaklassen repräsentiert. Sie bilden damit eine mögliche Ausprägung des Metamodells, die auch als Metamodellexemplar bezeichnet wird. Ein kleiner Ausschnitt aus dem Metamodell von FUJABA ist in Abbildung 2.13 dargestellt. Die Basisklasse für alle Modellelemente ist die Klasse Component. Aus dieser Klasse werden alle Objekte direkt oder indirekt abgeleitet. Die Basis für Klassendiagramme und Klassendiagrammelemente wird durch die Klasse UMLClass hergestellt. Die Basis für Zustandsautomaten wird durch die Klasse UMLRealtimeStatechart hergestellt. Diese Klassen stehen über die Assoziation in Beziehung zueinander, da ein Diagramm verschiedene Diagrammelemente enthalten kann. Eine Reihe von Klassen wie Port, Connector und Interface schaffen die Möglichkeit Kommunikationsschnittstellen zu modellieren.

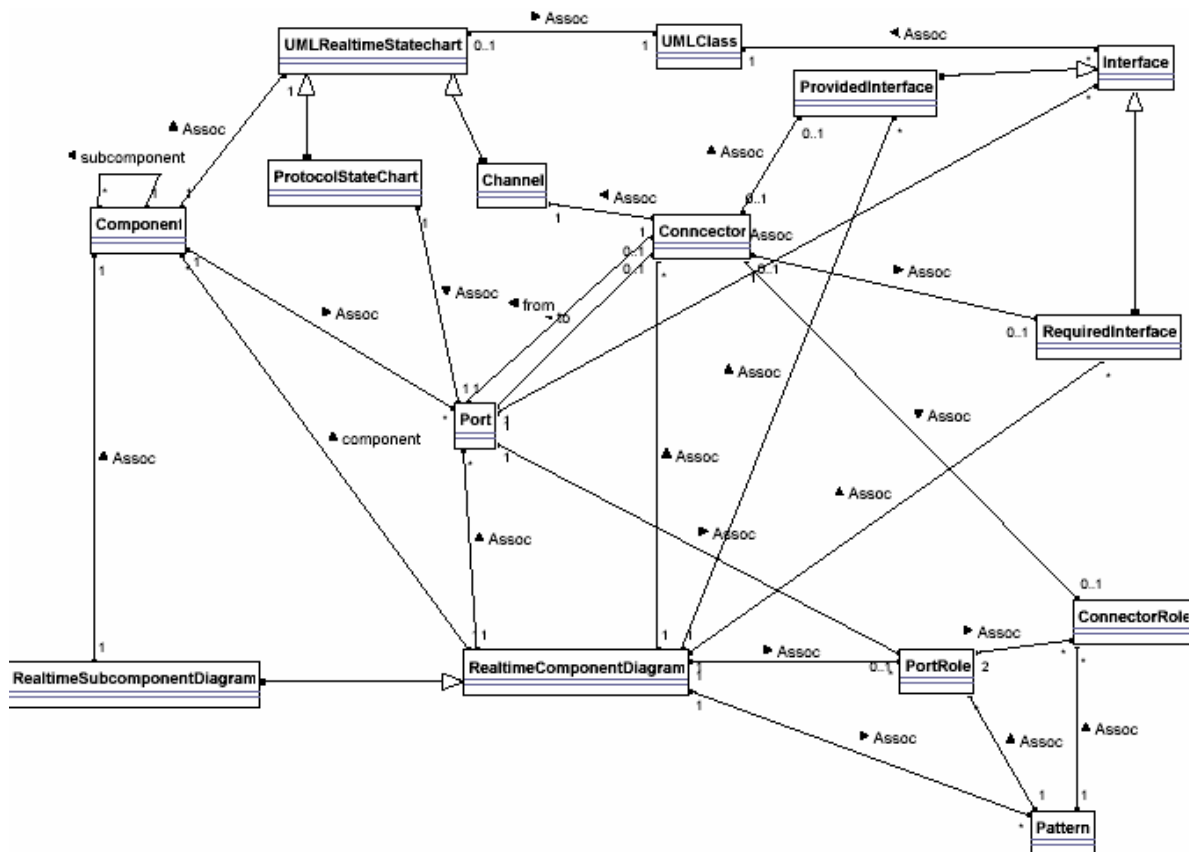


Abbildung 2.13.: FUJABA-Metamodell des RT-Component Diagramms

3. Lösungskonzepte

3.1. Erweiterung des bestehenden Modellierungskonzepts

Wie bereits in Kapitel 2.2 beschrieben, wurden die zur automatisierten Synthese von Schnittstellen benötigten Informationen in Form eines XML Schemas modelliert, welches den Namen IFS-Format trägt. Eine Instanz des IFS-Formats (IFS-Instanz) stellt als mögliche Ausprägung des XML Schemas eine gültige Beschreibung eines Kommunikationssystems dar, welche als Systemarchitektur bezeichnet wird. Abbildung 3.1 veranschaulicht dies in dem mit XML bezeichneten Kreisausschnitt. Die Aufteilung innerhalb der Grafik ist so gewählt, dass der orange Ring jeweils die domänenspezifische Modellierungssprache angibt. In blau dargestellt sind die entsprechenden Modelle, die in der spezifischen Sprache beschrieben sind. Ein solches Modell dient wiederum als Metamodell für die Bildung einer Modellinstanz, hier in gelb dargestellt. Die Modellebene (blauer Ring) ist durchgängig, da dort eine identische Information modelliert wird und somit eine Modelltransformation möglich ist. Als eine spezielle Ausprägung des Modells gilt dies folglich auch für die Instanzebene (gelber Ring). Um die Synthese zu automatisieren und die erdachten Konzepte zu evaluieren wurde der IFS-Editor als Java Werkzeug realisiert. Weiterhin ermöglicht der IFS-Editor die Visualisierung und Editierung von IFS-Instanzen. Unterschiedliche Sichten innerhalb des IFS-Editors repräsentieren dabei ausgezeichnete Teil-Schema des IFS-Formats. Die Implementierung des Editors basiert auf dem MVC (Model-View-Controller) Prinzip, d.h. man trennt Datenmodell, GUI (Graphical User Interface) und die darauf angewendeten Algorithmen. Ein Teil des objektorientierten Datenmodells, welches im Weiteren als IFS-Datenstruktur bezeichnet wird, implementiert exakt das IFS-Schema (blauer Pfeil), so dass IFS-Instanzen mühelos gespeichert und wieder geladen werden können (blauer Doppelpfeil). Die Repräsentation einer IFS-Instanz in Java wird als IFS-Objekt bezeichnet (vgl. Abbildung 3.1). Auf diesen Objekten wird die automatisierte Synthese ausgeführt.

Um eine modellbasierte Modellierung zu ermöglichen, wird das bisher bestehende Modellierungskonzept in dieser Arbeit auf UML2.0 ausgeweitet. Dazu muss das in Kap 2.2 beschriebene IFS-Format in ein UML2.0 Modell umgesetzt werden (roter Pfeil). Stattdessen könnte auch die bestehende IFS-Datenstruktur in ein UML2.0 Modell umgesetzt werden, wie dies mit diversen Reengineering Werkzeugen, wie z.B. Together oder Fujaba, möglich wäre. Das resultierende UML2.0 Modell wird als IFS-Modell bezeichnet, eine Instanz davon als IFS-Modell-Instanz (siehe Abbildung 3.1). Wie oben bereits erläutert ist ein Datenaustausch (Transformation) zwischen den verschiedenen Instanzen durch die Äquivalenz der Modelle bezüglich der beinhalteten Informationen problemlos möglich. Mögliche Konzepte und Realisierungsmöglichkeiten für die Transformation zwischen UML und dem bestehenden XML und Java werden nachfolgend betrachtet.

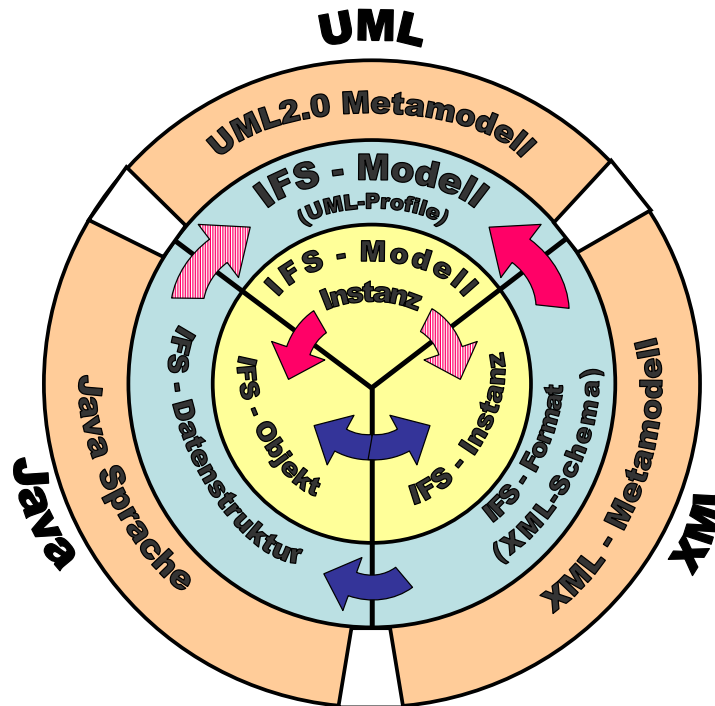


Abbildung 3.1.: Das um UML2.0 erweiterte Modellierungskonzept

3.2. Das Modellierungskonzept in UML2.0

Aufgrund des umfangreichen Sprachumfangs von UML2.0 ist es möglich die gesamte Information des IFS-Formats zu modellieren. Allerdings können die Modellierung und die sich anschließende Transformation des resultierenden Modells in das IFS-Format nur gelingen, wenn jedem Teil des IFS-Modells eine präzise Semantik bezogen auf das IFS-Format zugrunde liegt. Aus diesem Grund wurde ein UML Profile für das IFS-Modell definiert, das eine spezielle Semantik sowie gewisse Syntaxeinschränkungen beinhaltet. Eine ausführliche Beschreibung dazu wird in Kapitel 4 vorgestellt. Da das IFS-Schema auf einer einfachen XML Struktur basiert, ließe es sich ohne Weiteres durch ein großes Klassendiagramm ausdrücken, was in Abbildung 3.2 unten angedeutet ist. Auf diese Weise wäre das Modellieren für einen Designer sehr aufwendig, sofern die homogene Struktur von Klassendiagrammen nicht geeignet ist, um intuitiv Struktur- und Verhaltensaspekte gleichzeitig zu beschreiben. Das liegt darin begründet, dass die Klassen selbst keine weitere Semantik haben.

Aus diesem Grund wurde das IFS-Schema analysiert. Bei der Untersuchung waren zwei Fragen zu beantworten: Welche Diagrammtypen in UML2.0 können sinnvoll Teile des IFS-Formats ausdrücken und wie kann man diese zu einer objektorientierten modellbasierten und intuitiven Modellierung zusammenstellen? Die Untersuchung hat ergeben, dass zusätzlich zu den Klassendiagrammen das Komponentendiagramm und Zustandsautomaten verwendet werden um eine optimale Modellierung zu gewährleisten (vgl. Abbildung 3.2). Die Diagrammtypen werden für die Modellierung wie folgt eingesetzt: Die Struktur der Systemarchitektur wird durch Komponentendiagramme modelliert. Unter Struktur versteht man die Hierarchie der Architekturkomponenten und deren Verbindungen untereinander. Das Klassendiagramm legt fest, wie die

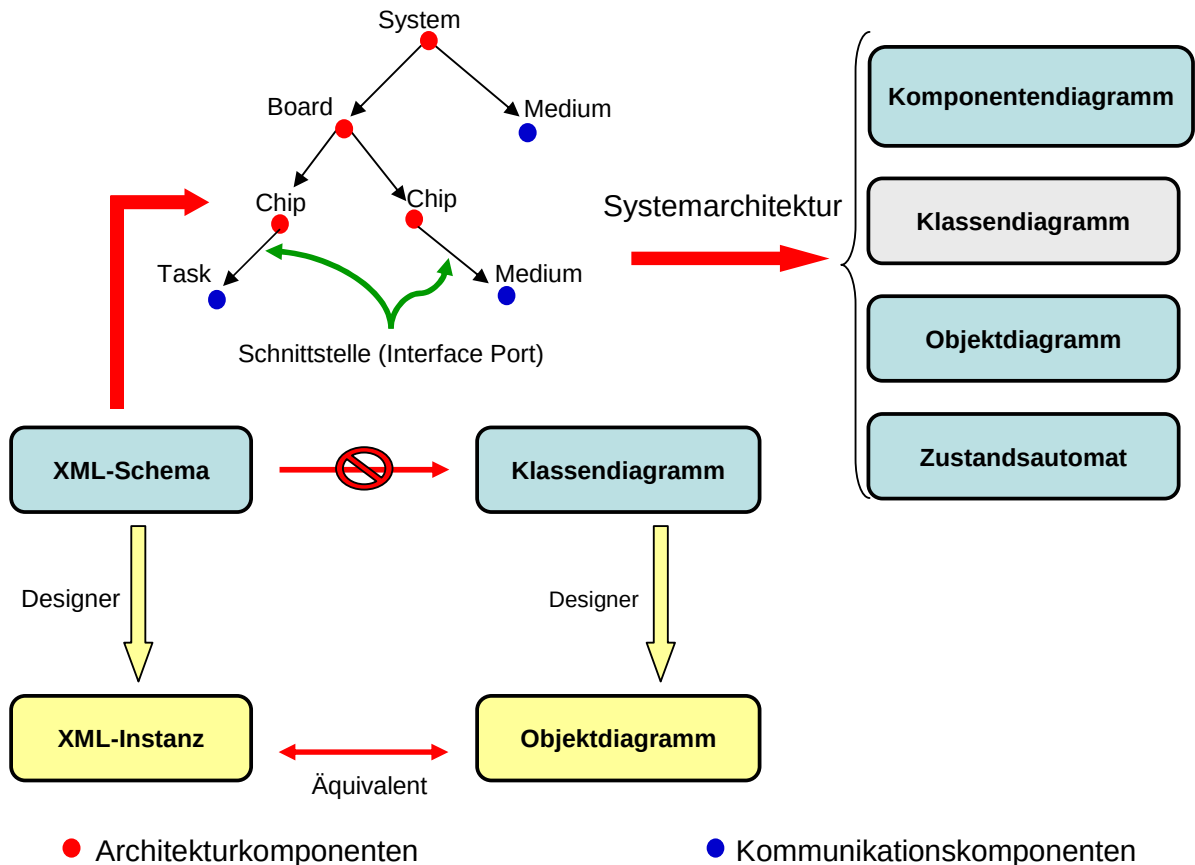


Abbildung 3.2.: Das Modellierungskonzept in UML2.0

statischen Eigenschaften der Systemarchitektur in Form von Attributen und Datentypen bei der Instanziierung in den Objektdiagrammen verwendet werden. Die Verhaltensbeschreibung, welche sich auf die Beschreibung von Kommunikationsprotokollen bezieht, wird durch Zustandsautomaten realisiert. Wie in Abbildung 3.2) zu sehen ist, wird das IFS-Schema als IFS-Modell in UML2.0 definiert. Das definierte UML Profil ist inhaltlich und semantisch auf das IFS-Modell beschränkt und dient dem Designer als Metamodell um IFS-Modell-Instanzen zu generieren. Die detaillierte Aufteilung der Systemarchitektur und ihre Repräsentation im UML2.0 Modell, die Erweiterung des UML-Metamodell durch das Profil, sowie die Attributdeklaration der einzelnen Klassen werden in Kapitel 4 erläutert. Ebenfalls wird dort für jeden Teilaspekt der Systemarchitektur eine Repräsentation als UML2.0 Modell vorgelegt.

3.3. Modelltransformation und Lösungsansätze

Die Transformation beschreibt die Abbildung des UML2.0 Modells in das bereits bestehende IFS-Format. Mit Hilfe eines UML2.0 Editors werden IFS-Modell-Instanzen modelliert (siehe Abbildung 3.3). Die resultierende IFS-Modell-Instanz soll dann in ein IFS-Objekt bzw. eine IFS-Instanz transformiert werden (vgl. Abbildung 3.1). Für den Fall, dass kein direkter Zugriff auf die Datenstruktur der modellierten IFS-Modell-Instanz existiert, muss eine Transformation

über ein Austauschformat durchgeführt werden. Der Abschnitt 3.3.1 stellt mögliche Formate für die Transformation vor. In dem Abschnitt 3.3.2 werden Lösungsansätze für die Transformation näher ausgeführt.

3.3.1. Modelltransformation

Die meisten UML Tools ermöglichen die Exportierung von UML Modellen in einem XMI[XML] Format und unterstützen die Importierung von XMI Dateien. XML (eXtensible Markup Language) ist ein Standard zur Erstellung strukturierter, maschinen- und menschenlesbarer Dateien [Wik]. Sofern das XML nicht objektorientiert ist, gibt es multiple Wege für eine Abbildung von Modellen. Das hat zur Folge, dass die Interoperabilität zwischen Werkzeugen auf dieser Ebene nicht mehr gewährleistet werden kann. Mit Hilfe der Standarddarstellung des XMI(XML Metadata Interchange) lassen sich Modelle durch XML darstellen, wodurch unterschiedliche Entwicklungswerkzeuge ihre Objekte untereinander austauschen können. Die Flexibilität des XMI Dateiformats erlaubt ihren Einsatz in unterschiedlichen Werkzeugen. Für diese Arbeit ergeben sich daraus zwei Varianten die Transformation durchzuführen. Bei der ersten Variante würde die resultierende IFS-Modell-Instanz als XMI-Datei gespeichert und müsste dann erneut geladen werden, um die Transformation durchzuführen (vgl. Abbildung 3.3, rot schraffierte Pfeile).

Aus der XMI-Datei wird mit Hilfe von XSLT, Dom oder SAX die entsprechende IFS-Instanz generiert, welche dann als XML-Datei gespeichert werden kann. Die zweite Variante besteht in der direkten Umsetzung der Datenstruktur im Speicher durch einen entsprechenden Übersetzer (vgl. Abbildung 3.3, roter Pfeil). Dazu benötigt man allerdings den Programmcode, der im allgemeinen Fall bei kommerziellen Werkzeugen nicht zur Verfügung steht. Die beiden Lösungsansätze werden nachfolgend näher erläutert.

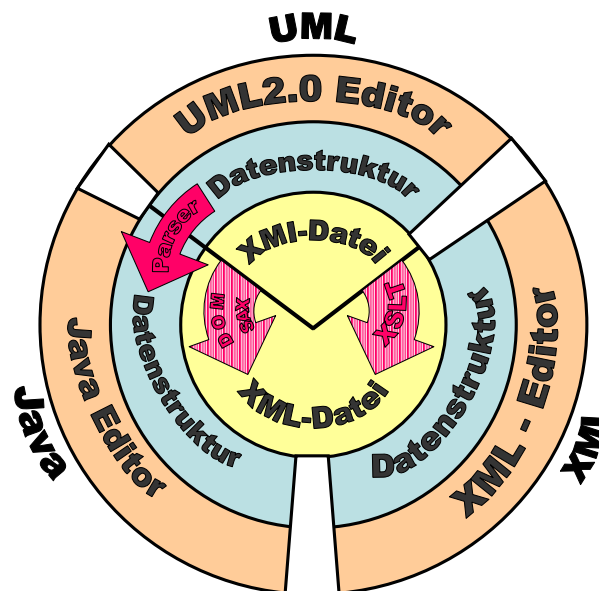


Abbildung 3.3.: Modelltransformation

3.3.2. Lösungsansätze

Es gibt verschiedene Ansätze, die die Bearbeitung von XML-Datei ermöglichen. Drei Verfahren werden beispielhaft vorgestellt. Danach wird auf das Parsen von Datenstrukturen im Speicher eingegangen.

XML Parser

- **SAX [Bre01]**

SAX (Simple API for XML) ist eine standardisierte Möglichkeit um XML Dateien zu bearbeiten. Das Programm kann allerdings nicht in einem XML Dokument hin- und herspringen und kann nur in einem beschränkten Anwendungsgebiet eingesetzt werden. SAX ist sequentiell. Die Informationen über ein XML Dokument stehen nur in dem Moment des Parsens zur Verfügung und gehen verloren, sobald der Parser fortschreitet. Mit einer selbst geschriebenen Parser-Callback-Methode können Daten, Elemente, Attribute und Strukturen im zu parsenden Dokument manipuliert werden. Ein Vorteil von SAX ist, dass beim Parsen nur ein kleiner Teil der XML Datei im Speicher sein muss und sich damit die Geschwindigkeit erhöht. Dies ist aber ein Nachteil, wenn man mehrere Informationen in der ganzen Datei für die Bearbeitung benötigt, da man selbst entsprechende Datenstrukturen zur Erfassung und Erhaltung der Daten im Speicher entwickeln muss. Aus diesem Grund ist es sehr aufwendig und nicht zweckmäßig diesen Ansatz zum Exportierten von UML Modellen das IFS-Schemas zu verwenden. Ein Beispielcode befindet sich im Anhang unter A.2.1

- **DOM [Bre01]**

DOM (Document Object Model) ist die zweite Variante, um XML Dateien auszuwerten. DOM ist eine in Java implementiertes Konzept mit definierten Methoden, die das Lesen, Anlegen und Modifizieren von Objekten innerhalb von Dokumenten ermöglichen. Sie stellt im Gegensatz zu SAX eine Baum-Repräsentation der Objekte zur Verfügung. Dies geschieht, indem die gesamte XML Datei gelesen wird und benötigt daher viel Speicherplatz. Der Vorteil bei diesem Ansatz ist, dass alle Elemente der XML Datei in einer hierarchischen Struktur vorliegen und Zugriffe darauf mit Hilfe der objektorientierten Datenstruktur erfolgen. Auch dieser Ansatz wurde in dieser Arbeit nicht weiter verfolgt, da es neben dem hohen Speicherbedarf auch recht aufwendig ist DOM Dokumente zu tranversieren. Hinzu kommt, dass hier XML bzw. XMI Dateien geladen (DOM) und gespeichert (UML-Tool und DOM) werden müssen. Das Beispiel in Anhang unter A.2.2 zeigt, wie ein DOM-Parser-Programm geschrieben werden kann, um XML Dokumente einzulesen.

- **XSLT [Bre01]**

Einen weiteren Ansatz bietet XSLT (Extensible Stylesheet Language Transformation). XSLT ist eine Programmiersprache zur Transformation von XML-Dokumenten. Sie erlaubt die Definition von Umwandlungsregeln von einem XML Quellformat auf beliebige andere Zielformate. Nachdem die Stylesheet-Regeln für die Transformation definiert wurden, ist jeder XSLT-Prozessor in der Lage, den XSLT-Stylesheet zu lesen und eine XML-Datei in das gewünschte Ausgabeformat zu transformieren. Als Erläuterung wurde mit

3. Lösungskonzepte

Hilfe des UML Werkzeuges Together 6.1 ein Klassendiagramm mit dem Namen System und zwei Attributen modelliert(siehe Abbildung 3.4. Das Modell wurde als XMI-Datei gespeichert.

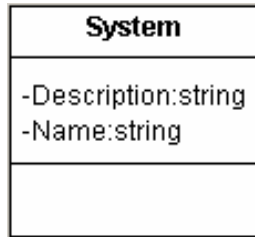


Abbildung 3.4.: Die Klasse “System”

Das Exportieren von einem Modell in eine XMI-Datei lässt sich nur durch zusätzliche Speichierungen von Informationen realisieren. Diese Informationen dienen der Identifikation, Referenzierung von Elementen oder beinhalten grafische Koordinaten. Das Beispiel 3.1 repräsentiert den für das Beispiel relevanten Teil der gespeicherten XMI-Datei.

Beispiel 3.1: XML-Beispieldatei

```
<?xml version = '1.0' encoding = 'Cp1252' ?> <XMI xmi.version = '1.1'
xmlns:UML = '//org.omg/UML/1.3'>
<XMI.header>
</XMI.header>
<XMI.content>
  <UML:Class xmi.id = 'S.7' name = 'System'
    visibility = 'public' isSpecification = 'false'
    isAbstract = 'false' isActive = 'false'>
    <UML:Attribute xmi.id = 'S.8'
      name = 'Description' visibility = 'private'
      isSpecification = 'false'
      changeability = 'changeable' ownerScope = 'instance'>
    </UML:Attribute>
    <UML:Attribute xmi.id = 'S.9'
      name = 'Name' visibility = 'private'
      isSpecification = 'false' changeability = 'changeable'
      ownerScope = 'instance'>
    </UML:Attribute>
  </UML:Class>
</XMI.content>
</XMI>
```

Dieses Dokument definiert sich selbst als XML Version 1.0. Das Beispiel 3.1 repräsentiert das dazu gehörende XSL Stylesheet, der das bestehende XML Format auf das neue Format in Beispiel 3.2 umwandelt. Bei der Umwandlung in das Zielformat werden nicht relevante Informationen einfach verworfen. Die Struktur des XST Stylesheets legt dann fest, wie das Ausgabeformat aussehen soll.

Beispiel 3.2: Das Sytlesheet zu Beispiel 3.1

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:template match="UML:Class">
  <xsl:variable name="TypeName_Sys" select="name()" />
  <xsl:if test="$TypeName_Sys=_System">
    <System>
      <Identification>
        <xsl:template match="UML:Attribute">
          <xsl:variable name="TypeName_Name" select="name()" />
          <xsl:if test="$TypeName_Name=_Name">
            <Name>
              <xsl:value-of select="@name" />
            </Name>
          </xsl:if>
          <ID>1</ID>
          <xsl:variable name="TypeName_Desc" select="name()" />
          <xsl:if test="$TypeName_Desc=_Description">
            <Description>
              <xsl:value-of select="@name" />
            </Description>
          </xsl:if>
          <xsl:apply-templates />
        </Identification>
      </System>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Beispiel 3.3: Die resultierende IFS-Instanz

```
<?xml version="1.0" encoding="UTF-8"?>
<System>
  <Identification>
    <Name>Name</Name>
    <ID>1</ID>
    <Description>Description</Description>
  </Identification>
</System>
```

Parsen der Datenstruktur im Speicher

Die gesamte Datenstruktur liegt hier bereits im Speicher vor. Das hat zum Vorteil, dass auf die modellierten Daten schnell im Hauptspeicher zugegriffen werden kann. Damit erhöht sich die Geschwindigkeit bei der Transformation. Das IFS-Modell muss nicht mehr erst als Datei gespeichert und dann geladen werden, bevor es bearbeitet wird. Allerdings hat der Parser der Datenstruktur den Nachteil, dass der entwickelte Algorithmus abhängig von dem eingesetzten UML Werkzeug ist.

Die resultierende IFS-Modell-Instanz der Systemarchitektur wird mit einem geeigneten Parseralgorithmus geparkt und eine IFS-Instanz bzw. IFS-Objekt erzeugt, das dann mit Hilfe des IFS-Editors geladen wird. Der Parseralgorithmus wird in Kapitel fünf beschrieben.

4. Modellierung

4.1. Grundgedanken

Dieses Kapitel beschreibt die Modellierung der Systemarchitektur mit den beinhalteten Schnittstellenbeschreibungen (IFD) und den Implementierungsplattformbeschreibungen (TPD) des IFS-Formats in UML2.0. Wie bereits in Kapitel 2.2.2 erwähnt wurde, lassen sich Kommunikationssysteme durch ihre Struktur und ihr Verhalten charakterisieren. Diese Unterteilung in Struktur und Verhalten, d.h. in eine physikalisch-elektrische Sicht und das Kommunikationsprotokoll werden bei der Modellierung berücksichtigt. Die Struktur der Systemarchitektur ebenso wie ihre statischen Eigenschaften werden durch Komponentendiagramme unter Zuhilfenahme von Objektdiagrammen modelliert. Die Modellierung des dynamischen Verhaltens erfolgt anhand von Zustandsautomaten. Abschnitt 4.2 erläutert diesen Aspekt. Anhand der hier vorgestellten Modellierung wird die Semantik des UML2.0 Profils basierend auf dem IFS-Modell definiert.

4.2. Aufbau des Modells

Im Rahmen der UML2.0 Erweiterungsechanismen wurde das IFS-Profil definiert, das die Beschreibung von IFS-Modellen ermöglicht. Durch das Stereotypenkonzept das im Kapitel 2.1.4 vorgestellt wurde, werden die Klassen *“IFSKomponente”*, *“IFSInterface”*, *“IFSPort”*, *“IFSConstraint”* und *“IFSActivity”* definiert, um die entsprechenden Klassen Component, Interface, Port, Constraint und Activity aus dem Metamodell zu erweitern. Die IFSKomponente-Klasse hat eine Vererbungsbeziehung zur UML Komponent-Klasse(Component), und definiert wie Komponenten im IFS-Kontext zu interpretieren sind. Alle Elemente der Systemarchitektur sind als Stereotype definiert. Die Abbildung 4.1 repräsentiert den Aufbau der Systemarchitektur als Klassendiagramm. Die Systemarchitektur basiert auf IFSKomponenten, die eine Identifikation und eine Version enthalten. Die Identifikation sowie die Version werden nachfolgend erläutert. Wie in Kapitel 2.2 erwähnt, lassen sich die IFSKomponenten in zwei Gruppen aufteilen. Wie in Abbildung 4.1 zu sehen ist, sind das die Kommunikations- (rot) und Architekturkomponenten (blau), die zur Strukturbeschreibung der Systemarchitektur gehören.

Identifikation

Die Identifikationsklasse hat Attribute, die das eindeutige Identifizieren und Referenzieren von einzelnen Systemarchitekturkomponenten ermöglichen. Das sind folgende Attribute, die in Abbildung 4.2 dargestellt sind:

- **Name:**

Mit diesem Attribut wird einer IFSKomponente ein vom Designer lesbarer Name vergeben.

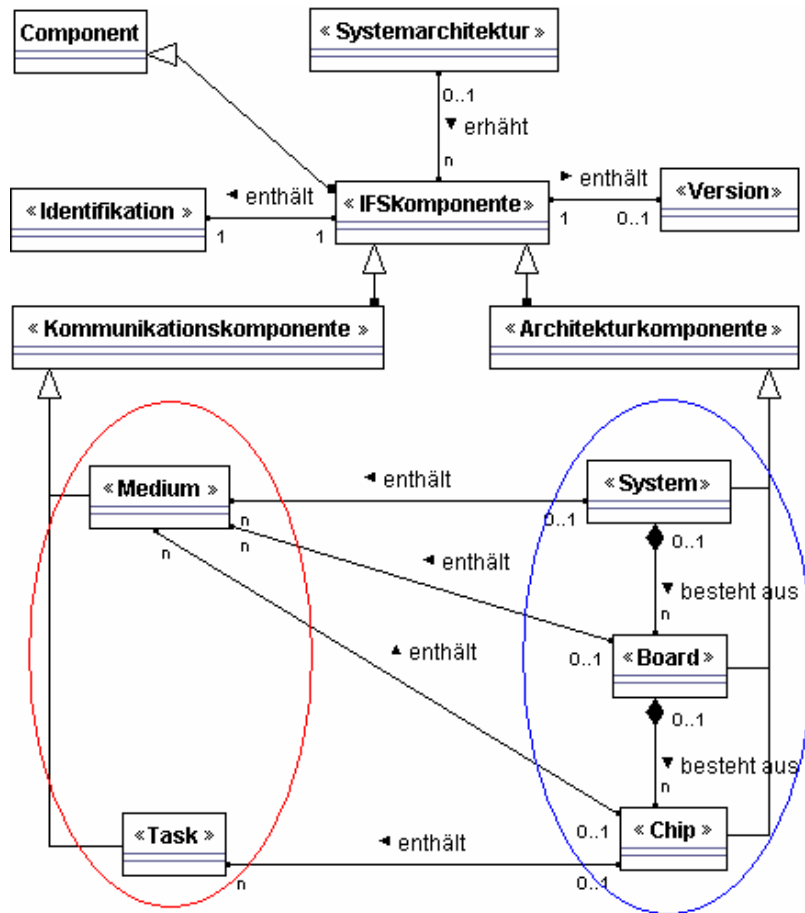


Abbildung 4.1.: Klassendiagramm der Systemarchitektur

- **Description:**

Anmerkungen und Kommentare zu IFSKomponenten werden mit Hilfe des Attributes Description beschrieben. Es besteht die Möglichkeit, Descriptions als UML-Kommentar im Klassendiagramm einzugeben.

- **ID:**

Die IFSKomponenten und Teile der IFSKomponenten, die innerhalb des IFS-Schemas vorkommen, haben eine lokal eindeutige ID zur internen Identifizierung. Sie werden bei der Modellierung nicht berücksichtigt und erst bei der Transformation des Modells hinzugefügt. Im UML-Modell werden Objekte durch ihre Namen eindeutig gekennzeichnet.

- **Category:**

Das Kategorieattribut ordnet ihren IFSKomponenten einen bestimmten Typ zu. Das IFS-Format bietet mittels der Technik der Enumerations, je nachdem, um welche IFS-Komponente es sich handelt, verschiedene Typen zur Auswahl an [Fic03]. Eine Liste der zur Auswahl stehenden Enumerations findet sich im Anhang A.1.2(siehe Abbildung A.13, A.14, A.15, A.16). Der entsprechende Typ muss bei der Instanziierung des Klassendiagramms eingegeben werden. Die Systemklasse hat kein Kategorieattribut.

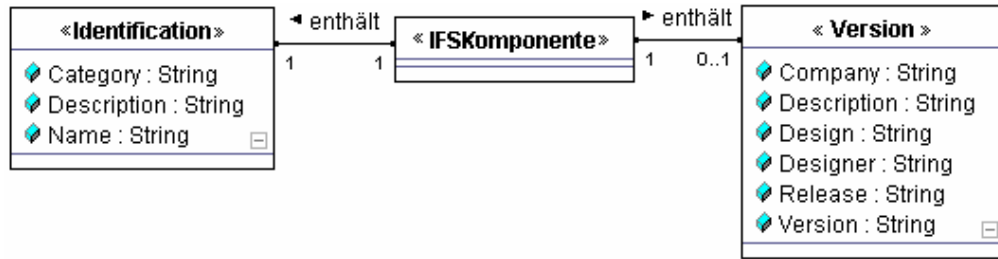


Abbildung 4.2.: Die Klasse “IFSKomponente”

Version

Die Versionsklasse besitzt Attribute zur Eingabe von Versionsinformationen. Das sind die Namen der Company, des Designs und des Designers, sowie die Description für Anmerkungen und Kommentare. Die Release- und Versionsattribute beziehen sich auf Angaben zu den Archiven.

4.2.1. Strukturbeschreibung

Die Kommunikations- und Architekturkomponenten werden mit Hilfe des Komponentendiagramms modelliert. Die Hierarchie der Systemarchitektur wird bei der Modellierung berücksichtigt. Das System steht auf der obersten Ebene der Strukturbeschreibung der Systemarchitektur gefolgt von Board, Chip, Task und Medium. Medien können prinzipiell auf allen Hierarchieebenen vorkommen. Die unterste Ebene der Strukturbeschreibung bezieht sich auf die Tasks. Die jeweilige Kommunikation zwischen IFSKomponenten wird über die Stereotypen IFSPort der dazwischen liegenden Hierarchieebenen realisiert. IFSPort ist eine Erweiterung der UML-Metamodellklasse Port (siehe Abbildung 4.3). Für die Kommunikation sind Interface-

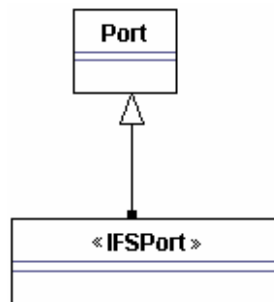


Abbildung 4.3.: Stereotype “IFSPort”

klassen erforderlich. Das UML Metamodell ermöglicht es, mehrere Interfaces für einen Port zu definieren (siehe Abbildung 2.4). Allerdings wird bei der Modellierung des IFS-Modells immer nur ein Interface betrachtet. Die Interfaces unterscheiden sich zwischen required und provided Interfaces. Mit dieser Unterscheidung wird die Richtung der Kommunikation bestimmt. Eine Verbindung zwischen zwei Interfaces, der so genannte Konnektor, kann nur durch zwei Interfaces mit unterschiedlichem Typ realisiert werden (siehe Abbildung 2.3).

4. Modellierung

Die physikalischen und elektrischen Eigenschaften der Schnittstelle werden als Attribute in den IFSInterface-Stereotypen abgelegt (siehe Abbildung 4.4). Diese Attribute haben bestimmte Wertbereiche, die bei der Implementierung des IFSInterfaces einzuhalten sind. Die Wertbereiche dieser Attribute befinden sich im Anhang (siehe Abbildung A.19). Der IFSInterface-Stereotyp ist ebenso wie IFSPort eine Erweiterung des Interfaces des UML-Metamodells. Die für die Kommunikation benötigten Signale werden als Stereotyp definiert und stehen in einer Assoziations-Beziehung mit ihrer IFSInterface-Klasse. Die Attribute dieser Signale werden bei der Erzeugung einer IFS-Modell-Instanz mit konkreten Werten gefüllt. Das Attribut SignalName enthält den Namen des Signals z.B. Start. Das Attribut Value gibt den Initialwert eines Signals an und legt dadurch implizit auch die Bitbreite des Signal fest, z.B. Value = "000" bzw. Value = '000'. Aus diesen Signalen werden bei der Modelltransformation die physikalischen Pins der Interfaces, sowie die logischen Protokollpins, auf denen das Protokoll spezifiziert wird, abgeleitet.

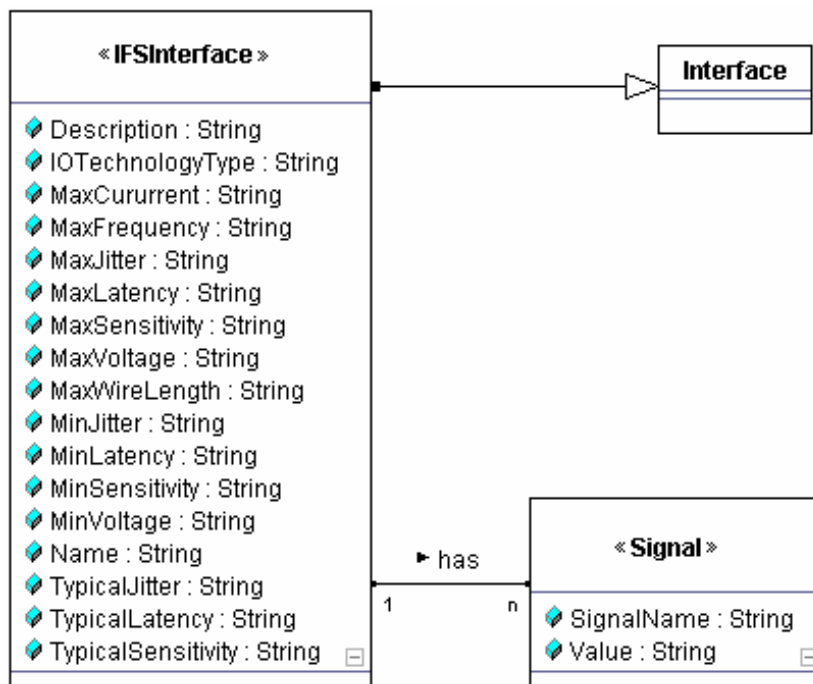


Abbildung 4.4.: Die Klasse “Interface”

Die Abbildung 4.5 präsentiert beispielhaft ein Kommunikationssystem “*sys1*” mit zwei Boards “*b1*” und “*b2*”, die jeweils einen Chip haben. Der Chip “*chip1*” beinhaltet einen Task “*t1*” und ein Medium “*m2*”. Das Board “*b1*” beinhaltet weiterhin zusätzlich zur dem Chip “*chip1*” ein Medium “*m1*”. Der Chip “*chip2*” beinhaltet einen Task “*t2*”. Die jeweilige Kommunikation zwischen den IFSKomponenten wird über die IFSPorts der eingeschlossenen Hierarchieebenen geleitet. Hier nicht sichtbar sind die benötigten IFSInterfaces mit ihren deklarierten Signalen.

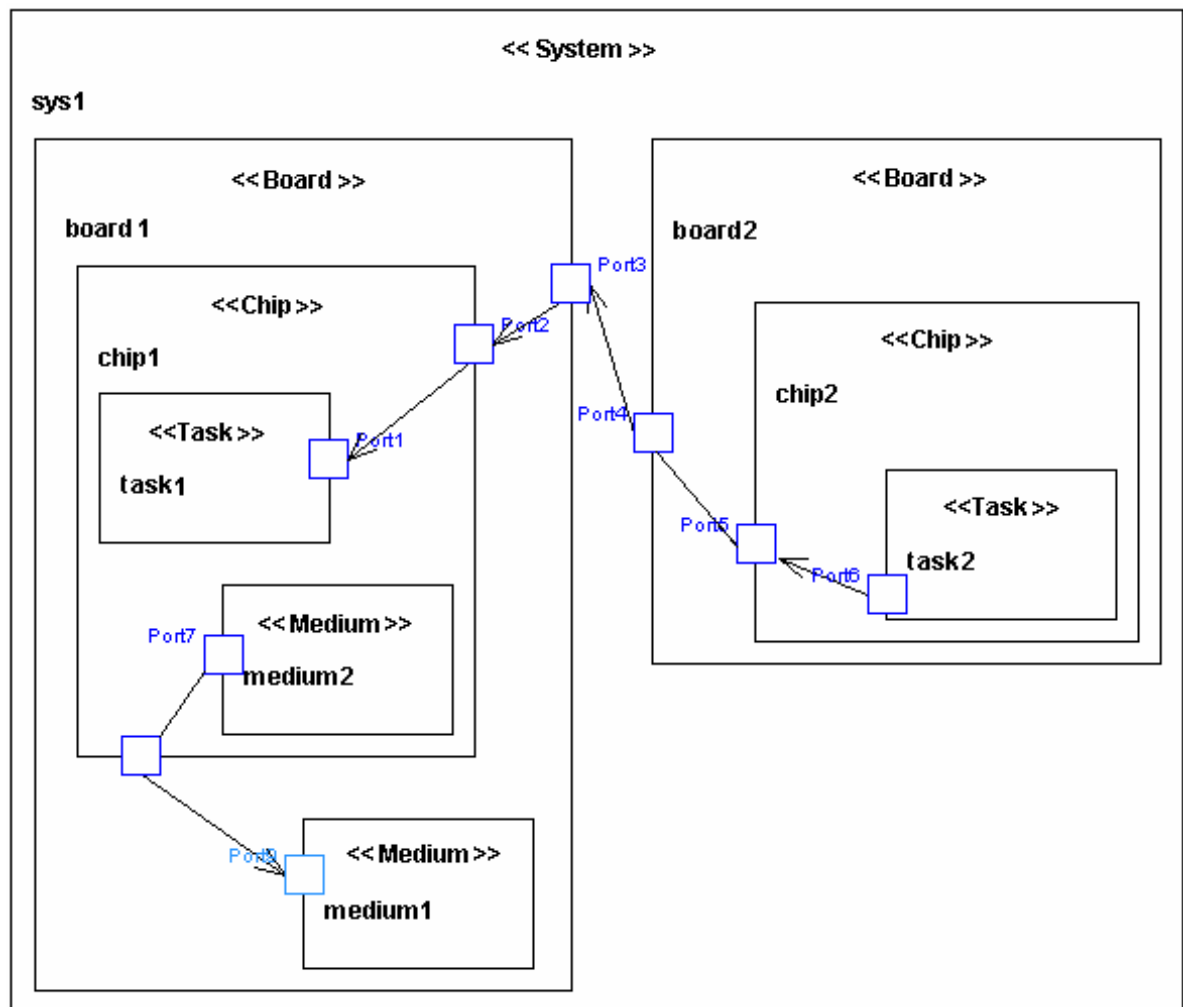


Abbildung 4.5.: Die Systemarchitektur als Komponentendiagramm

4.2.2. Verhaltensbeschreibung

Die Kommunikationskomponenten implementieren die Funktionalität des Gesamtsystems. Aus diesem Grund besitzen die Kommunikationskomponenten im Gegensatz zu den Architekturkomponenten neben dem IFSInterface auch ein Protokoll. Dabei handelt es sich um die Beschreibung des Kommunikationsverhaltens der Kommunikationskomponente. Das innere Verhalten von Task und Medium wird als “Black Box” Modell betrachtet. Wie Abbildung 4.6 darstellt, ermöglicht das UML-Metamodell, dass Ports eine ProtocolStateMachine referenzieren. Die ProtocolStateMachine erbt dabei von der State-Machine-Klasse. Das Protokoll der Kommunikationskomponenten wird in der zugehörigen ProtocolStateMachine ihres IFSPorts modelliert.

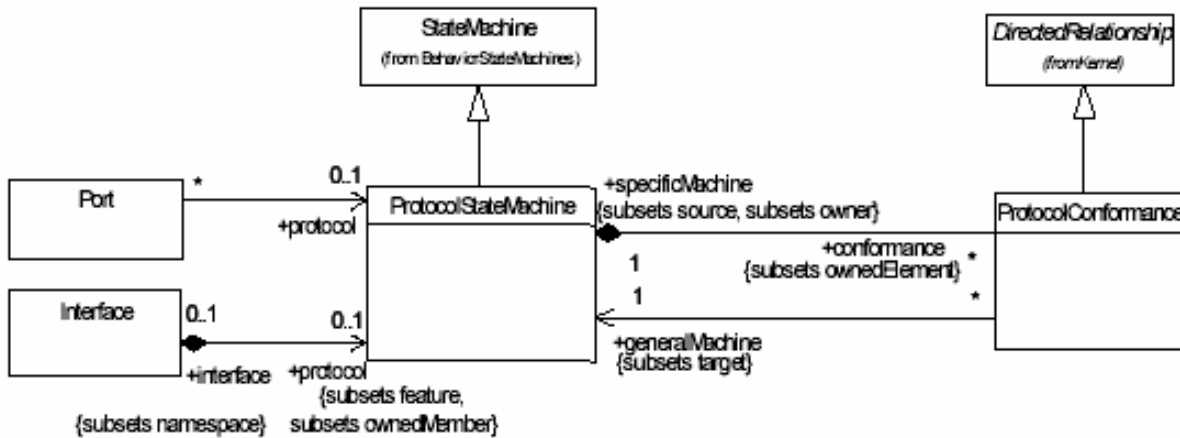


Abbildung 4.6.: UML Protocol state machines [Uni04]

Die vollständige Modellierung des Protokolls (siehe Abbildung 4.7), wie es im IFS-Format vorkommt, wird durch das UML-Metamodell der ProtocolStateMachine nicht ermöglicht. Aus diesem Grund wurde die Protokollbeschreibung, wie in Abbildung 4.6 gezeigt, in zwei Gruppen aufgeteilt. Die erste Gruppe beinhaltet die rot eingekreisten Klassen, welche gemäß UML Metamodell als Zustandsautomat modelliert werden. Die restlichen Klassen gehören zu der zweiten Gruppe, die außerhalb des Zustandsautomaten modelliert werden. Darunter finden sich die Klassen ReferenceSignal sowie die Attribute der Klasse Protokoll und deren Identifikation. Dazu zählt noch die Klasse zur Beschreibung der Implementierungsplattform (Target Platform Description, kurz TPD). Die einzelnen Klassen der beiden Gruppen werden im Folgenden durch das UML-Metamodell beschrieben.

ProtocoloPins

Wie bereits in Abschnitt 4.2.1 erwähnt wurde, sind die ProtocolPins virtuelle Pins. Die Werte der ProtocolPins werden in den einzelnen Zuständen des Protokolls als Automatenausgabe des Zustandsautomaten spezifiziert. Unter bestimmten Voraussetzungen können die Werte von ProtocolPins auch als Bedingung für einen Zustandsübergang angegeben werden. Durch die virtuellen ProtocolPins können Protokolle unabhängig von der physikalischen Struktur, auf der sie ausgeführt werden, modelliert werden. In einer Kommunikationskomponente werden die ProtocolPins des eingesetzten Protokolls dann an die physikalischen Pins der IFSInterfaces gemapped. Da diese Trennung in UML nicht vorgesehen ist, werden die ProtocolPins des IFS-Modells direkt aus den physikalischen Pins des jeweiligen IFSInterfaces abgeleitet.

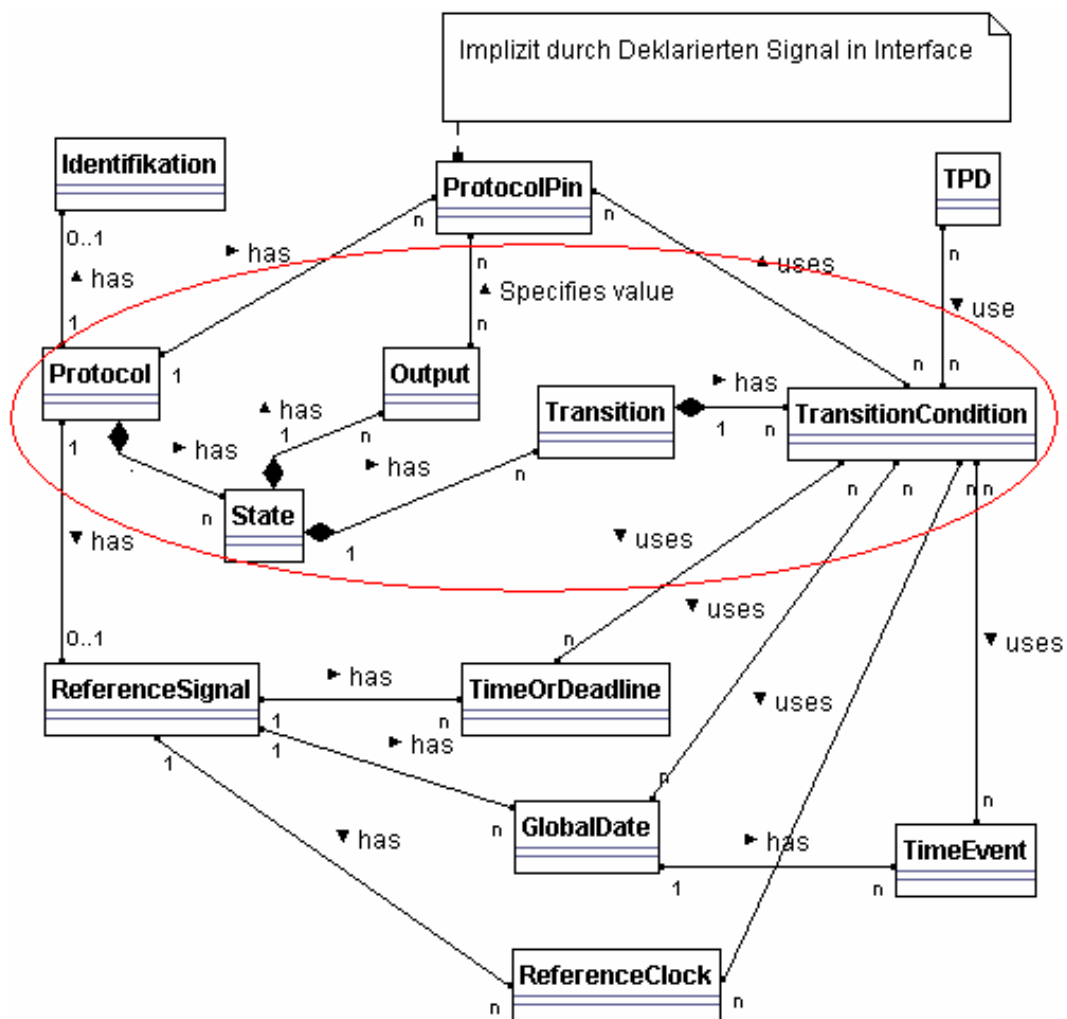


Abbildung 4.7.: Aufbau des Kommunikationsprotokolls des IFS-Schemas als Klassendiagramm

Der Zustandsautomat

An dieser Stelle ist es wichtig zu erwähnen, dass bei dieser im IFS Umfeld nur Mooreautomaten betrachtet werden. Das I/O-Verhalten einer Kommunikationskomponente wird als endlicher Zustandsautomat (FSM) innerhalb der jeweiligen ProtocolStateMachine des zugehörigen IFS-Ports beschrieben. Nur die Kommunikationskomponenten haben Protokolle, daher werden nur die ProtocolStateMachines ihrer IFSPorts betrachtet. Die IFSPorts der Architekturkomponenten werden ignoriert, sofern es sich um die Beschreibung des Protokollverhaltens handelt. Es wird angenommen, dass die IFSPorts der Architekturkomponenten das Verhalten der verbundenen Kommunikationskomponenten “erben”.

Ein wichtiger Aspekt der Protokolle sind die Zustände mit deren Zustandsübergängen und Übergangsbedingungen sowie die (zustandgebundenen) Ausgaben, die in Abbildung 4.7 mit Hilfe des roten Kreises gekennzeichnet sind. Sie werden durch Zustandsautomaten modelliert. Das Protokoll für die Kommunikation unterliegt gewissen Einschränkungen und Eigenschaften,

4. Modellierung

die bei der Transformation der IFS-Modell-Instanz ermöglichen, die Protokollinformationen der IFS-Instanz zu konstruieren. Abbildung 4.8 zeigt das Klassendiagramm des Zustandsautomaten im UML-Metamodell. Die Klassen *Constraint* und *Activity* aus dem Metamodell wurden durch Stereotype *IFSConstraint* und *IFSActivity* erweitert. Die benötigten Protokollinformationen des IFS-Formats sind nachfolgend als Attribute der IFSConstraint-Klasse und der IFSActivity-Klasse angegeben.

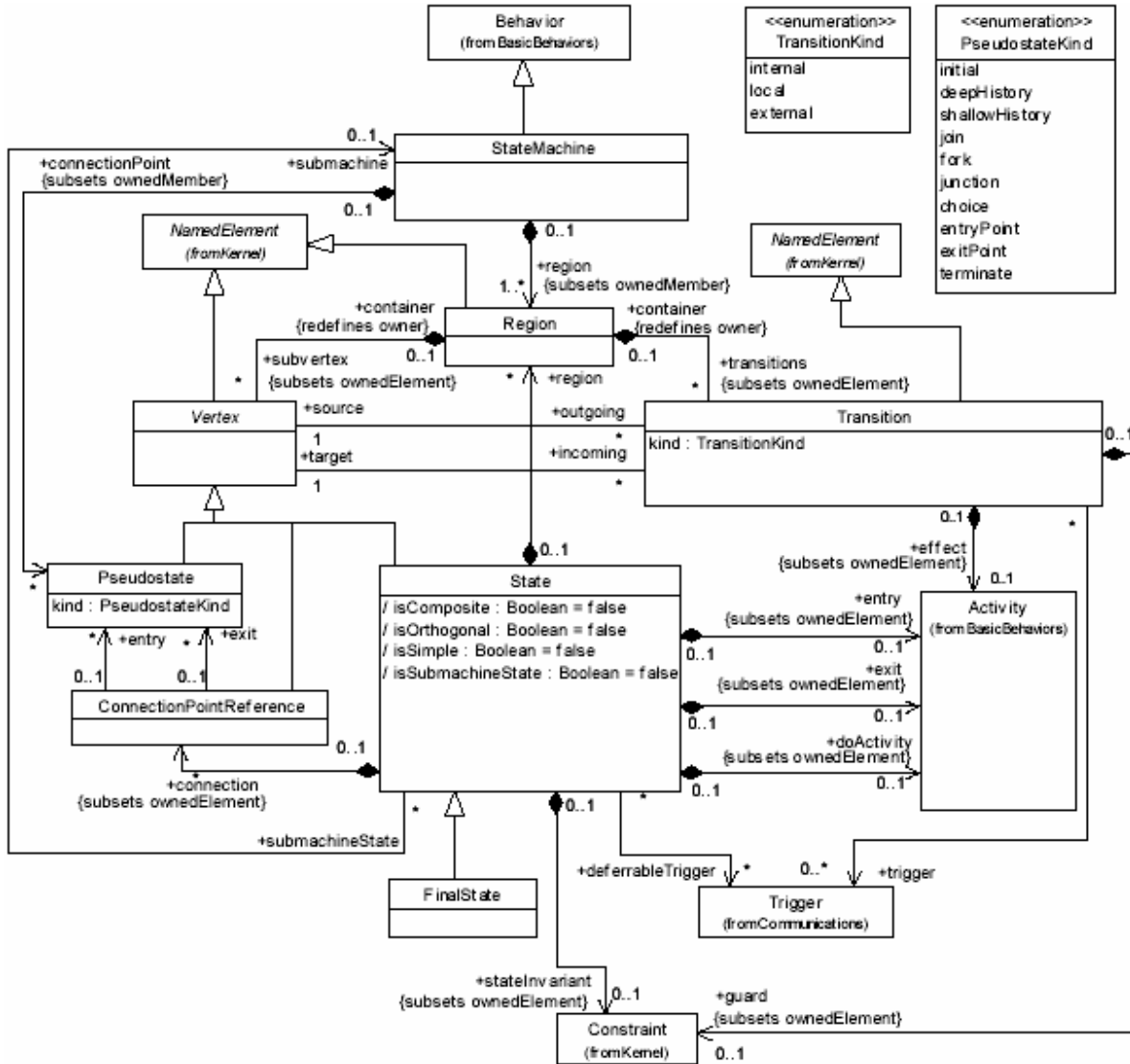


Abbildung 4.8.: Klassendiagramm eines Zustandsautomaten aus dem UML2.0 Metamodell [Uni04]

IFSConstraint-Klasse

In der IFSConstraint-Klasse werden die notwendigen Parameter zur Beschreibung von Zustandsübergangsbedingungen als Attribute abgelegt (siehe Abbildung 4.9). Diese Attribute repräsentieren getriggerte Signale, deren Werte bestimmte Bedingungen erfüllen müssen, um den

Zustandsübergang anzustoßen. Die möglichen Werte, die bei der Modellierung benutzt werden können, stehen als Kommentar im Klassendiagramm. Allerdings werden bei der Modellierung die im Kommentar angegebenen Abkürzungen verwendet.

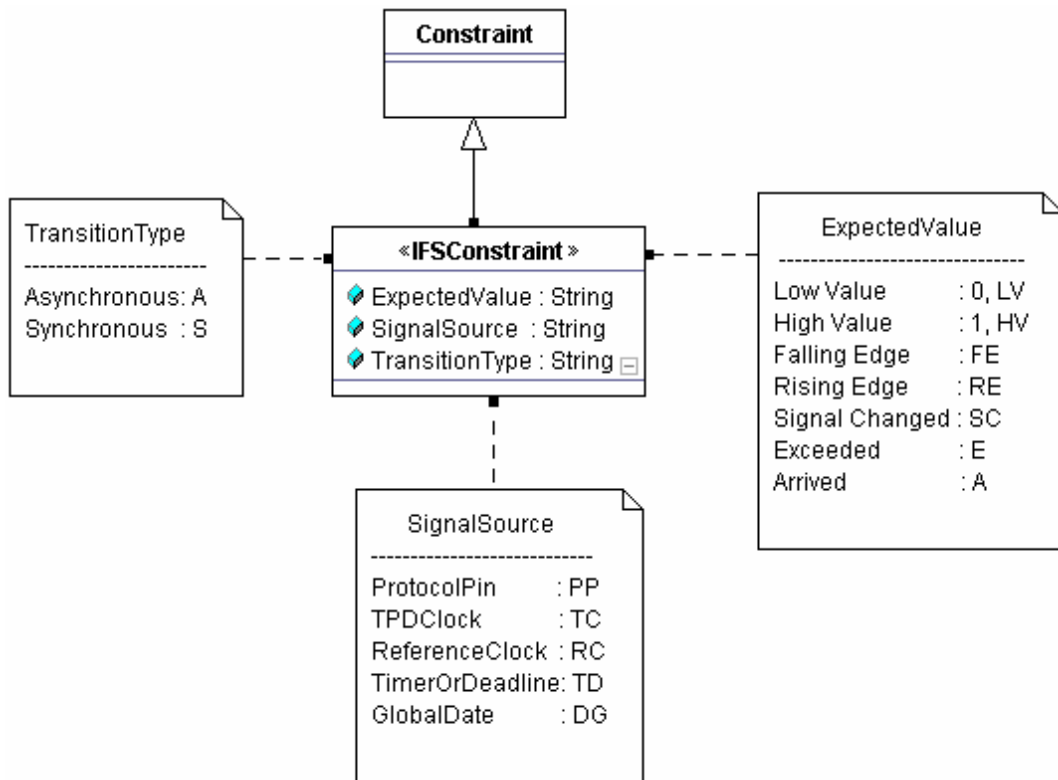


Abbildung 4.9.: Die Klasse “IFSCConstraint”

- **SignalSource:**

Die *SignalSource* legt fest, welcher Signaltyp für den *Trigger* benutzt wird. Ob es sich um eine *ProtocolPin*, eine *TPDCKlock*, eine *ReferenceClock*, eine *TimerOrDeadline* oder ein *GlobalDate* handelt. Die Klassen *TPDCKlock*, *ReferenceClock*, *TimerOrDeadline* und *GlobalDate* wie in Abbildung 4.7 angegeben, werden als Unterpunkt der *TPD* bzw. der *ReferenceSignals* genauer erläutert.

- **TransitionType:**

Der *TransitionType* drückt aus, ob es sich um ein asynchrones oder synchrones Signal handelt.

- **ExpectedValue:**

Mit diesem Attribut werden die erwarteten Werte angegeben, die den Zustandsübergang auslösen. Mögliche Werte sind *Low Value*, *High Value*, *Falling Edge*, *Rising Edge* und *Signal Changed*. Falls es sich um eine *ProtocolPin* handelt, wird direkt der Wert des Signals übernommen z.B. *ExpectedValue* = “101101”. Ein *TimerOrDeadline* nutzt immer den Wert *exceeded*. *GlobalDate* erwartet den Parameter *arrived*.

IFSActivity-Klasse

Die IFSActivity-Klasse beinhaltet die Informationen über die Automatenausgaben, die der modellierte Automat in einem bestimmten Zustand (*State*) liefert (siehe Abbildung 4.10). Dort werden die Werte aller *ProtocolPins* (Wert der Signale) im aktuellen Zustand spezifiziert. Mit dem Attribut *Behavior* wird das Verhalten von *ProtocolPins* angegeben. Die möglichen Werte sind *Data* und *Control*. Das ist erforderlich, da sich in seriellen Protokollen die Bedeutung einzelner Signale über die Zustände hinweg ändern kann. Das Attribut *Direction* spezifiziert die Richtung, in der die Daten übertragen werden. Hier sind Input und Output vorgesehen.

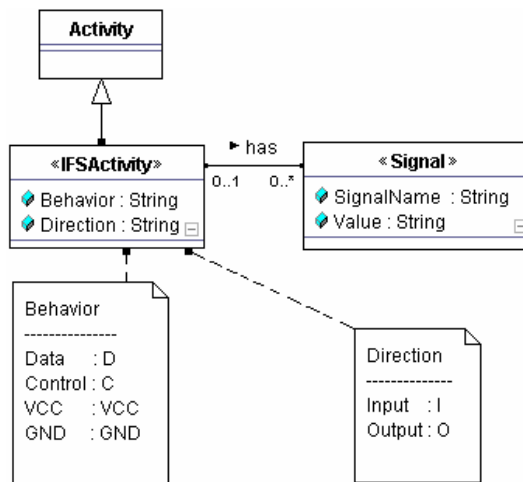


Abbildung 4.10.: Die Klasse “IFSActivity”

State

Das UML-Metamodell der State-Klasse wurde nicht erweitert. Ein aussagekräftiger Name für jeden Zustand ist hier erwünscht. Der Name ermöglicht die Identifikation einzelner Zustände, zusätzlich wird er für die Erzeugung der IFS-Instanz verwendet.

Trigger

FPGA sind synchrone Architekturen die mit Clocks arbeiten. Asynchrone Ereignisse müssen daher auf synchronisiert werden. Daraus folgt, dass bei der Modellierung des Zustandsautomaten die Trigger aus dem UML-Metamodell nicht genutzt werden und man sich auf die Benutzung von IFSConstraints beschränken kann.

Signal

Der Signalname dient als Schlüssel zur Identifizierung eines Signals, im Gegensatz zum IFS-Format, wo IDs vergeben werden. Aus Sicht der *ProtokollPins* handelt es sich um den Namen eines deklarierten Signals in der IFSInterface-Instanz. Für die Stereotypen *TPDCKlock* sowie die *ReferenceClock* und die *TimerOrDeadlines* handelt es sich um den Namen der Instanz ihrer Klasse. Anhand dieses Namens lokalisiert der Parseralgorithmus später die instanziierte Klasse in der Systemebene. Ein *GlobalDate* wird dagegen anders interpretiert. Falls *GlobalDate* als *SignalSource* ausgewählt wurde, wird für den Signalnamen das zugehörige *TimeEvent* ausgewählt.

ReferenceSignal

ReferenceSignals sind ein wichtiger Teil der Protokollbeschreibung, da hier verschiedene Typen von Zeitmodellen beschrieben werden können. Die Klasse *ReferenceSignals* ist in die drei Stereotypen *GlobalDate*, *TimerOrDeadline* und *ReferenceClock* aufgeteilt. Diese Stereotype haben eine Assoziationsbeziehung zum Systemstereotype (siehe Abbildung 4.11), und wurden bereits bei der Deklaration der *SignalSource* verwendet. Die Instanzen dieser Klassen werden auf der Systemebene abgelegt, so dass die Protokolle aller Kommunikationskomponenten die *ReferenceSignals* in den *TransitionConditions* als getriggertes Signal verwenden können (Abbildung 4.9 und Abbildung 4.11).

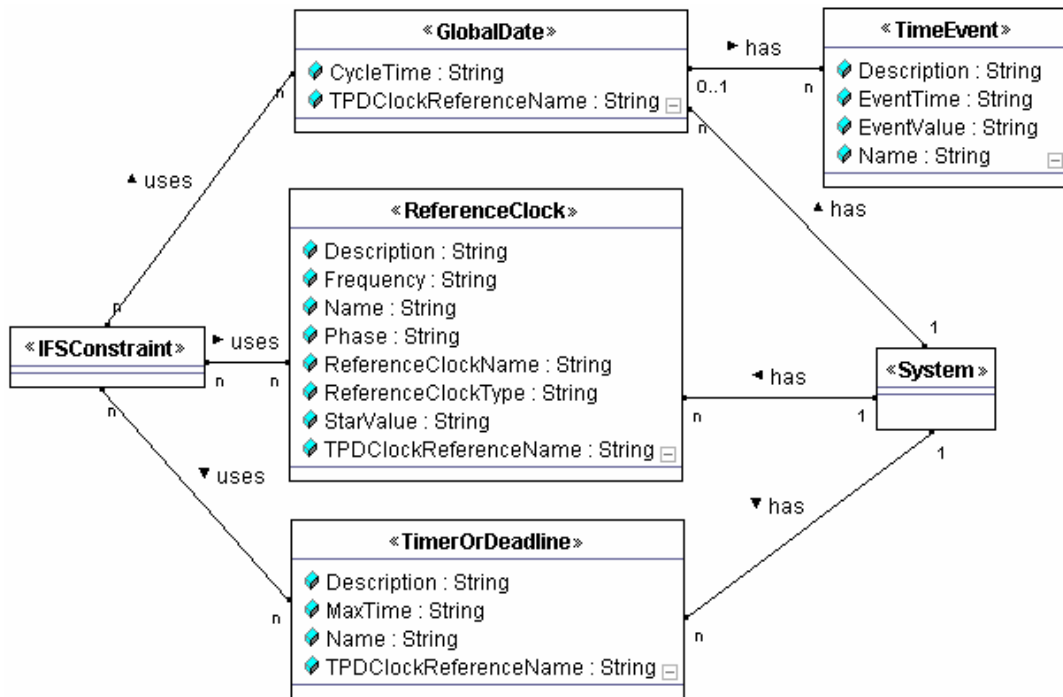


Abbildung 4.11.: Die Klasse "ReferenceSignal"

Der Stereotype *GlobalDate* wird benutzt, um periodisch auftretende Ereignisse zu beschreiben und besteht aus den folgenden Attributen:

- **CycleTime:**
Dieses Attribut ermöglicht es, die Periodendauer der zyklischen Zeitbasis zu definieren.
- **TPDClockReferenceName:**
Der Stereotype *GlobalDate* sowie *ReferenceClock* und *TimerOrDeadline* besitzen alle das Attribut *TPDClockReferenceName*. Mit Hilfe dieses Attributs wird die zugehörige *TPDClock* anhand ihres Namens auf der Chipebene identifiziert, aus der das entsprechende *ReferenceSignal* abgeleitet wird. Der Designer kann an dieser Stelle eine *TPDClock* referenzieren, die bereits im zugehörigen Chip deklariert ist.

4. Modellierung

Die *GlobalDates* haben zusätzlich ***TimeEvents*** die aus den folgenden Attributen bestehen:

- **Name und Description:**
Mit dem Attribut Name werden die einzelnen Signale identifiziert, denen jeweils eine Menge von Ereignissen zugeordnet wird. Ein Ereignis ist ein Tupel aus *EventTime* und *EventValue*. Signale können mit einer *Description* versehen werden.
- **EventTime**
Mit diesem Attribut wird der Zeitpunkt des Ereignisses innerhalb der *CycleTime* festgelegt.
- **EventValue**
Das Attribut *EventValue* legt das entsprechende Ereignis (*Low-* oder *High-Pegel*) fest, das zum Zeitpunkt *EventTime* am Signal des entsprechenden *GlobalDates* anliegen soll. Die möglichen Werte sind: *Logic One*, *Logic Zero*, *High Peak*, *Low Peak* und *Signal Change*.

ReferenceClocks sind künstliche Clocks, die jeweils aus einer der bestehenden Clocks abgeleitet werden. So können fast beliebige Taktraten für ein Kommunikationsprotokoll generiert werden. Die Attribute der Klasse *ReferenceClock* haben folgende Attribute:

- **Frequency:**
Mit diesem Attribut wird die Frequenz des *Referenzsignals* bestimmt, z.B. Frequency = 1.0E2 [Hz].
- **StartValue:**
Die Zuweisung eines Anfangswertes erfolgt mit Hilfe des Attributs *StartValue*: Mögliche Werte sind *Logic One* und *Logic Zero*.
- **ReferenceClockName:**
Der Designer kann an dieser Stelle die Clock referenzieren, die bereits in der zugehörigen *TPD* deklariert ist und aus der die Taktrate abgeleitet wird. Ein Clock-Fehler, der möglicherweise aus Rundungsfehlern resultiert, kann exakt berechnet werden.
- **ReferenceClockType:**
Der Designer kann mit Hilfe dieses Attributs festlegen, ob es sich um eine Clock der *TPD* oder einer anderen *ReferenzClock* handelt. Zur Auswahl sind daher entsprechend Real Clock (*TPD*) und *ReferenceClock* gegeben.
- **Phase:**
Diese Angabe gibt die Phasenverschiebung zu einer weiteren *ReferenzClock* an: 0° , 45° ($Pi / 4$), 90° ($Pi / 2$), 135° ($3 Pi / 4$) und 180° (Pi). Z.B. Phase = $Pi/4$.

Das *ReferenceSignal TimerOrDeadline* besteht nur aus einem Attribut. Ein *TimerOrDeadline* wird als Zähler implementiert. Ob der *TimerOrDeadline* als Timer oder Deadline interpretiert wird, hängt letztlich von seiner Anwendung im Protokoll ab.

- **MaxTime:**
Diese Angabe liefert die Zeitschranke des Zählers.

Target-Platform-Description

Der Stereotype *Target-Platform-Description* (*TPD*) wird auf der Chipebene modelliert. Die *TPD* hat zusätzlich zu der Assoziationsbeziehung zu dem Chipstereotype eine weitere Assoziationsbeziehung zu dem Stereotype *TPDClock* (siehe. Abbildung 4.12). In einer *TPD* werden die Eigenschaften eines Chips beschrieben, die charakteristisch für den Chip als Implementierungsplattform sind. Die *TPD-Klasse* findet bei der ReferenceSignal Deklaration Anwendung. Sie hat folgende Attribute für die Beschreibung von nutzbaren Ressourcen eines Chips:

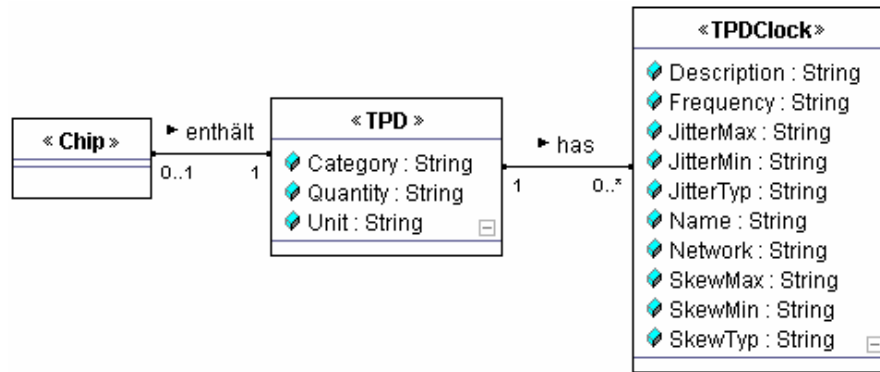


Abbildung 4.12.: TPD Klassendiagramm

- **Category:**
Die Einordnung eines Chips in die Kategorien: *FPGA*, *ASIC* oder *Processor* sind die möglichen Kategorien, die das IFS-Schema bisher unterstützt.
- **Quantity:**
Mit dem Attribut *Quantity* wird die Anzahl der verfügbaren Ressourcen angegeben.
- **Unit:**
Die Einheit, in der die Ressourcen angegeben werden, wird mit diesem Attribut angegeben. Zur Auswahl stehen *CLBCount*, *GateCount* und *Memory*.

TPDClocks haben, ähnlich wie *ReferenceClocks*, Attribute für die Beschreibung von Taktraten. Weiterhin sind Attribute für die Charakterisierung von Clock-Netzwerken vorhanden.

- **Frequency:**
Diese Attribute beschreibt die Frequenz der Clock.
- **Skew:**
Mit diesem Attribut wird der *Skew* der entsprechenden *Clock* angegeben.
- **Jitter:**
Gibt den *Jitter* der entsprechenden *Clock* an. Hier wird die maximale Abweichung vom Standardwert des Taktsignals angegeben.
- **Network:**
Mit diesem Attribut wird die technische Umsetzung des Clock-Netzwerks einer *Clock* auf dem Chip charakterisiert.

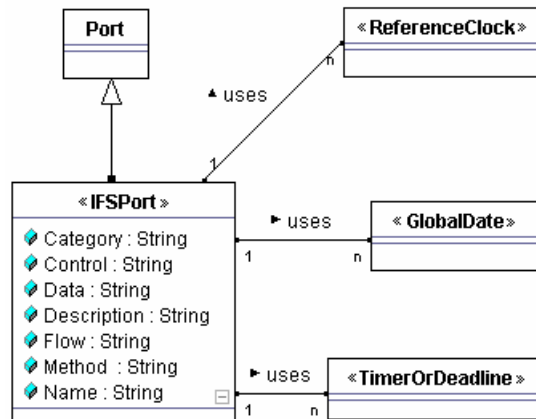


Abbildung 4.13.: Die Klasse “IFSPort”

Protokollattribute und Identifikationsattribute

Der Stereotype “*IFSPort*” ist eine Erweiterung der Portklasse des UML-Metamodells. Die Identification- sowie die Characterization-Attribute der Protokollklasse werden als Attribute im Stereotype “*IFSPort*” abgelegt (vgl. Abbildung 4.13). Die für die Protokollbeschreibung benötigten ReferenceSignals werden bei der Modellierung innerhalb des IFSPort-Instanz aufgeführt. Nur die GlobalDates, TimeEvents, TimerOrDeadlines und ReferenceClock Instanzen, die bei der Modellierung im Voraus auf der Systemebene deklariert wurden, können hier Anwendung finden (siehe Abbildung 4.11). Die Attribute Name, Description und Category sind die Identifikationsattribute. Folgende Typen stehen bis jetzt zur Auswahl bei der Instanziierung der Protokollkategorie: *RS232*, *RS485*, *LVDS*, *Firewire*, *UserDefined* und *Other*. Der Rest der Attribute sind die Characterization-Attribute. Diese wurden in das IFS-Schema integriert, um die Kompatibilität zum VSIA Format [Vir01] zu erfüllen und setzen sich aus folgenden Attributen zusammen:

- **Control:**
Mit diesem Attribut wird festgelegt, mit welcher IFSKomponente der Transaktionsprozess beginnt. Mögliche Werte für die Instanziierung sind *Responder*, *Initiator* und *Other*.
- **Data:**
Drei Typen *Consumer*, *Producer* und *Other* stehen hier zur Auswahl bei der Instanziierung dieses Attributs. Dadurch wird festgelegt, ob die zugehörige IFSKomponente Daten produziert oder annimmt.
- **Flow:**
Der Datenfluss des Protokolls wird mit Hilfe dieses Attributs bei der Instanziierung angegeben. Zur Auswahl stehen *Persistent*, *Buffered*, *FIFO*, *LIFO*, *Blocking*, *AssignedPortPriority*, *AssignedDataPriority*, *MultiRate*, *Pipelined*, *ExceptionsHandled* und *Other*. Folgende Werte sind im Moment realisiert *Persistent*, *AssignedPortPriority*, *Pipelined*
- **Method:**
Das Attribut Method kann folgende Werte bei der Instanziierung annehmen: *transRead*, *transWrite*, *messSense*, *messEmit*, *transOpenChannel*, *transCloseChannel*, *transSynchronize*, *transReset*, *transControl*, *messRead*, *messWrite* und *Other*.

5. Konzept und Implementierung der Modelltransformation

Das Modellierungskonzept sowie die verschiedenen Ansätze für die Modelltransformation sind bereits in Kapitel drei und vier unabhängig von einer konkreten Modellierungsumgebung und Implementierung vorgestellt worden. In diesem Kapitel wird die Transformation des IFS-Modells in die IFS-Datenstruktur vorgestellt. Damit können dann IFS-Modell-Instanzen(UML) in IFS-Objekte umgewandelt und durch die Unterstützung des IFS-Editors auch in IFS-Instanzen übersetzt werden. Zuerst wird in Abschnitt 5.1 eine Begründung für die Wahl der Transformationsart und Fujaba als UML Case-Tool geliefert. Dann wird in Kapitel 5.2 das für die Transformation angewendete Konzept vorgestellt. Anschließend wird in Abschnitt 5.3 die Implementierung des Transformationsalgorithmus näher erläutert.

5.1. Wahl der Transformationsart und des UML Case-Tools

Der Fujaba Editor ist ein Entwurf der Universität Paderborn. Der Programmcode liegt für Forschungszwecke vor. Auf dieser Basis hat sich ergeben, dass für die Arbeit des Parsens der Datenstruktur der Daten im Speicher Fujaba sehr gut geeignet ist. Das hat zum Vorteil, dass auf die Daten im Hauptspeicher schnell zugegriffen werden kann. Damit erhöht sich die Geschwindigkeit bei der Transformation. Die IFS-Modell-Instanz muss nicht mehr erst als XMI Datei gespeichert und dann geladen werden, bevor sie bearbeitet wird. Der andere Grund ist, dass der in Java geschriebene Programmcode es ermöglicht, eine direkte Transformation von einer objektorientierten Datenstruktur des Quell-Formats in eine andere objektorientierte Datenstruktur des Zielformats (IFS-Objekt) durchzuführen. Die Transformation der XMI Datei auf das Zielformat wurde mit hohem Aufwand erreicht und dadurch wenig Gewinn erbracht. Die gesamte Repräsentation des UML-Modells als XMI Datei ermöglicht es, einen allgemeinen Transformation Algorithmus zu generieren. Das Parsen der Datenstruktur im Speicher hat auch den Nachteil, dass der entwickelte Algorithmus abhängig von den Transformationstools ist, in diesem Fall von dem Fujaba Editor. Dies wäre nicht der Fall mit einer Transformation von einer XMI Datei.

Fujaba selbst unterstützt keine UML2.0 Profile. Von daher konnte das im Kapitel 4 vorgestellte Konzept der Erweiterung des UML-Metamodells nicht direkt eingesetzt werden. Das IFS-Profil wurde intern auf die bestehende Fujaba Datenstruktur abgebildet. Aus Sicht des Designers macht sich dies z.B. bei der Benennung der Stereotypen (siehe Abbildung 5.1) so wie bei der Beschreibung des Protokollautomaten (siehe Abbildung 5.2) bemerkbar.

Da die Protokollautomaten als Zustandsautomaten modelliert werden, können die Attribute der **IFSConstraint-Klasse** (siehe Abbildung 4.9) als Zustandsübergangsbedingungen mit folgender Syntax spezifiziert werden. Wie in Abbildung 5.2 in blauer Farbe dargestellt ist,



Abbildung 5.1.: Der Stereotype Board

wird zuerst der Transitionstyp (TransitionType), gefolgt von einem Punkt, angegeben. Danach kommen die Art der Signalquelle (SignalSource), ein Doppelpunkt und dann der Name des getriggerten Signals. Nach einem Gleichheitszeichen wird der erwartete Wert des getriggerten Signals (ExpectedValue) notiert. Die abstrakte Syntax dazu sieht wie folgt aus: TransitionType.SignalSource:SignalName = 'ExpectedValue'. Mehrere Zustandsübergangsbedingungen werden mit Hilfe von Klammern getrennt, um dann mit booleschen Operatoren verknüpft zu werden. Die booleschen Operatoren *AND*, *OR*, *NOR*, *XOR* und *NAND* stehen hier zur Auswahl.

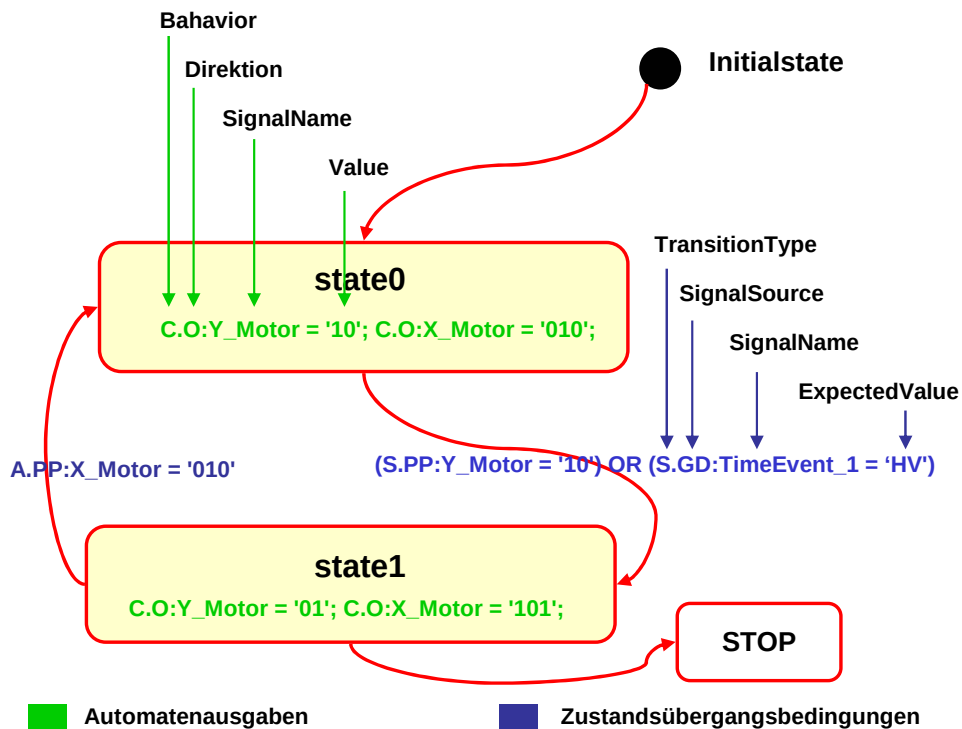


Abbildung 5.2.: Zustandsautomatenmodell

Die Attribute der **IFSActivity-Klasse** werden als Automatenausgaben mit folgender Syntax umgesetzt: Behavior. Direction:SignalName = 'Value';. Die einzelnen Ausgaben des Automaten werden mit Semikolons voneinander getrennt. Die Abbildung 5.2 zeigt ein Beispielmotell eines Zustandsautomaten mit einem Initialzustand, einem Stoppzustand und zwei weiteren Zuständen state0 und state1. Als Zustandsübergangsbedingungen und für die Zustandsausgaben wurden hier beispielhaft einige Deklarationen eingesetzt.

5.2. Transformationskonzept

Dieser Abschnitt dient dazu, das Transformationskonzept näher zu erläutern. Hier wird nur der für die Lösung der Aufgabenstellung verwendete Ansatz berücksichtigt. Die folgende Abbildung 5.3 stellt das Transformationskonzept dar. Die roten Pfeile repräsentieren die notwendigen Schritte, die für die Durchführung der Transformation durchgeführt werden. Der bestehende IFS-Editor dient als Ausgangspunkt für die Realisierung der Aufgabenstellung. Im IFS-Editor wird der Fujaba-Editor durch den Designer gestartet bzw. gestoppt (vgl. Abbildung 5.5 bzw. 5.6). Mit Hilfe des Fujaba-Editors werden IFS-Modell-Instanzen modelliert. Bei der Modellierung wird das in Kapitel 4 beschriebene Profil des IFS-Modells mit den Einschränkungen aus Kapitel 5.1 berücksichtigt. Die resultierende IFS-Modell-Instanz kann als XMI-Datei exportiert und wieder importiert werden oder mit den Eigenschaften des Werkzeugs als Fujaba-Projekt gespeichert und wieder geladen werden. Spätere eventuelle Änderungen an der IFS-Modell-Instanz (z.B. Änderung der Zustandsautomatenparameter), die durch den IFS-Editor mühsam durchgeführt werden, müssten somit entfallen.

Nachdem ein IFS-Modell modelliert wurde, kann das Modell in den IFS-Editor importiert werden. Dieser Prozess wird mit einem Knopf *“Import Fujaba”* im IFS-Editor gestartet (vgl. Abbildung 5.5). Die Importierung des IFS-Modells in den IFS-Editor geschieht mit Hilfe eines Parsers. Der Parser ist ein geschriebener Algorithmus, der die Quellsprache (IFS-Modell) und die Zielsprache (IFS-Format) kennt. Er generiert aus der entstehenden IFS-Modell-Instanz ein IFS-Objekt. Im nachfolgenden Abschnitt wird an entsprechender Stelle näher auf den Parser eingegangen. Die importierte IFS-Modell-Instanz kann mit Hilfe des IFS-Editors weiterverarbeitet werden. Es können z.B. durch die automatisierte Synthese IFB für inkompatible Schnittstellen generiert werden. An dieser Stelle sei erwähnt, dass der Import-Algorithmus zuerst ein IFS-Objekt aus dem IFS-Modell erzeugt, dieses dann als temporäre IFS-Instanz ablegt, welches der IFS-Editor dann erneut einlädt. Das ist deshalb sinnvoll, da der IFS-Editor bereits

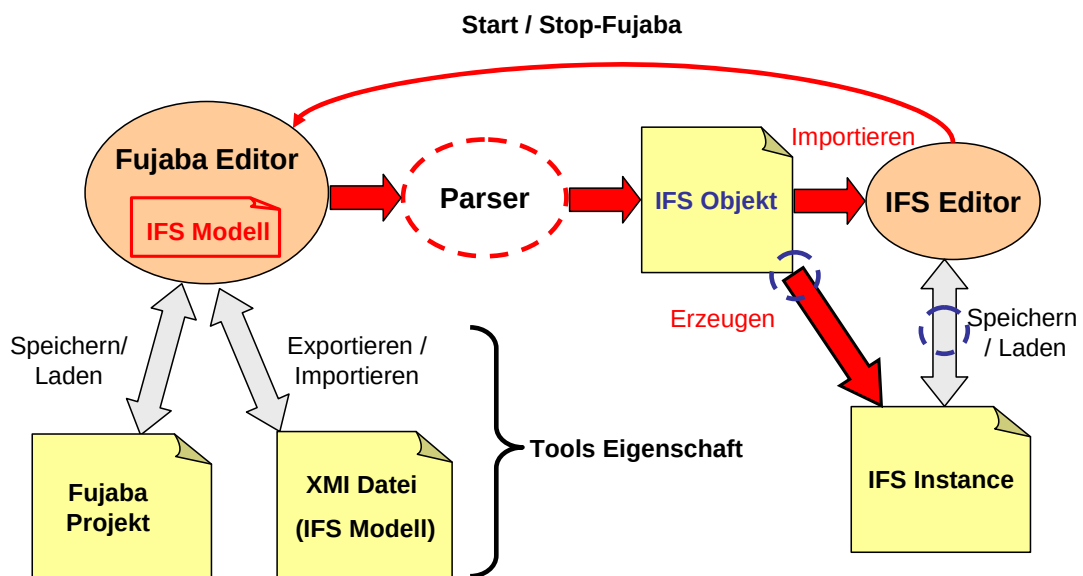


Abbildung 5.3.: Transformationskonzept

während des Ladens einer IFS-Instanz Algorithmen zum Aufbau seiner internen Datenstruktur auf den Daten ausführt. Diese Operationen können so sauber vom Import getrennt und im IFS-Editor belassen werden. Dieser Aspekt wird in Abbildung 5.3 durch einem blauen Kreis gekennzeichnet.

5.3. Implementierung der Modelltransformation

Nachdem nun das Transformationskonzept vorgestellt wurde, werden in diesem Abschnitt konkrete Techniken vorgestellt, die die Transformation der modellierten IFS-Modell-Instanz in die Zielsprache ermöglichen. Die im Rahmen dieser Arbeit entstandene Implementierung des Transformationskonzepts lässt sich durch das folgende Klassendiagramm beschreiben (vgl. Abbildung 5.4) und setzt sich aus den folgenden zwei Aspekten zusammen:

- Werkzeug-Kopplung
- Parseralgorithmus

Die IFS-Editor- und die Fujaba-Klasse sind komplexe Klassen mit unterschiedlichen Methoden und stehen in Relation mit weiteren Klassen für die Erfüllung ihrer Funktionalität. Die FujabaHandler- und die Synch2Fujaba-Klasse haben eine Assoziationsbeziehung mit allen Klassen der IFS-Datenstruktur. Die Systemarchitekturklasse ist ein Repräsentant der IFS-Datenstruktur und zeigt hier beispielhaft, wie die Klassen der IFS-Datenstruktur in Beziehung

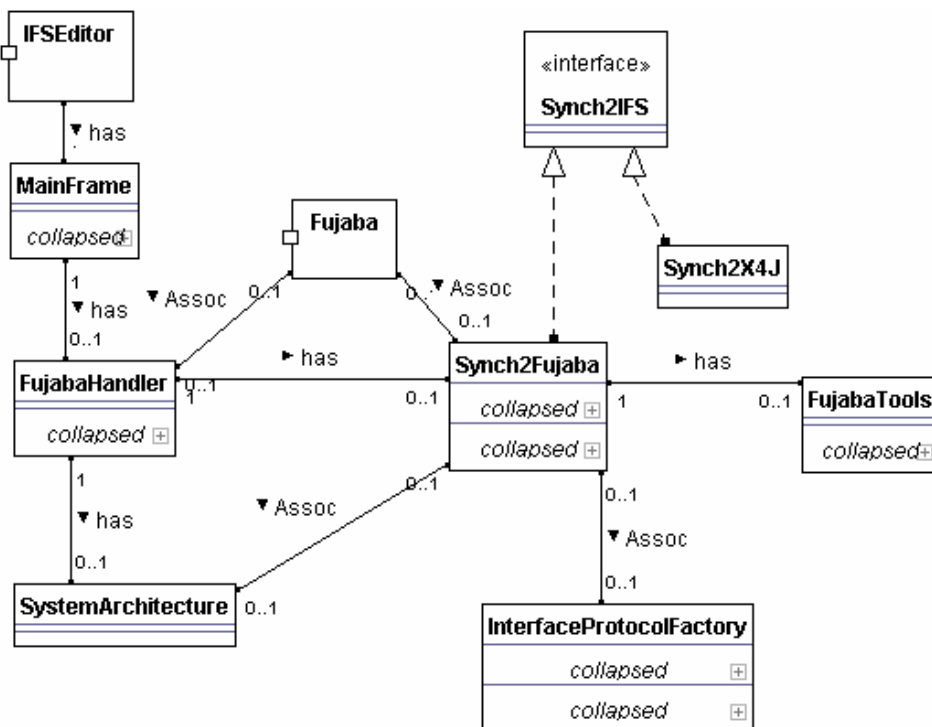


Abbildung 5.4.: Klassendiagramm der implementierten Modelltransformation

zu der FujabaHandler- und der Synch2Fujaba-Klasse stehen. Der eigentliche Parseralgorithmus wurde innerhalb der Synch2Fujaba-Klasse implementiert. Die Assoziation zu der Fujaba-Klasse ermöglicht es, zwischen den beiden Datenstrukturen zu navigieren um die Transformation durchzuführen. Das Interface Synch2IFS schreibt die erforderlichen Methoden für die Transformation in die IFS-Datenstruktur vor. Diese Methoden werden in der Synch2Fujaba- und der Synch2X4J-Klasse unterschiedlich implementiert. Das Interface wurde definiert, um die Entwicklung von Parsern für verschiedene Sprachen zu erleichtern, da das Ziel, die IFS-Datenstruktur zu erzeugen, immer das gleiche ist. Die Funktionalität dieser Klassen wird nachfolgend in den jeweiligen Unterpunkten näher erläutert.

5.3.1. Werkzeug-Kopplung

Der Fujaba-Editor wurde an den IFS-Editor angebunden. Der Fujaba-Editor wird per Klick auf den Knopf *Start Fujaba* des IFS-Editors gestartet (vgl. Abbildung 5.5). Nach dem Start (vgl. Abbildung 5.6) erscheint die Fujaba Oberfläche und die folgenden Operationen werden im IFS-Editor freigeschaltet. Der Start-Fujaba Knopf wird, sobald der Fujaba-Editor gestartet wurde, deaktiviert, wodurch nur die Möglichkeit einen Fujaba-Editor zu starten entsteht.



Abbildung 5.5.: IFS-Editor: Start Fujaba

Nach dem Start des Fujaba-Editors werden die Knöpfe *Stop Fujaba* und *import from Fujaba* aktiviert. Bei Klick auf den Knopf *Stop Fujaba* wird der gestartete Prozess des Fujaba-Editors beendet. Der Knopf *import from Fujaba* veranlasst die Importierung der aktuellen IFS-Modell-Instanz. Dazu muss in Fujaba-Editor ein gültiges Modell bestehend aus einem Komponenten-diagramm mit seinen zugehörigen Zustandsautomaten und Objektdiagrammen, wie in Kapitel 4 beschrieben, vorhanden sein. Ist das der Fall, so übersetzt ein Parseralgorithmus, der im Folgenden beschrieben wird, das UML Modell in die IFS-Datenstruktur und übergibt diese dem IFS-Editor.



Abbildung 5.6.: IFS-Editor: Import From und Stop Fujaba

MainFrame

In dem MainFrame des IFS-Editors wurden drei Methoden für die Darstellung des jeweiligen Knopfes (*Start*-, *Stop*- und *ImportFujaba* Button) eingefügt (vgl. Abbildung 5.7).

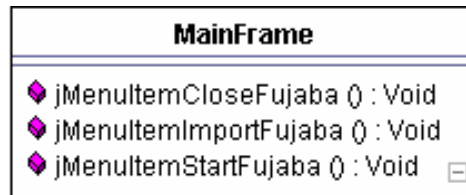


Abbildung 5.7.: MainFrame

FujabaHandler

Die Funktionalität der Knöpfe ist innerhalb der *FujabaHandler* implementiert (vgl. Abbildung 5.8). Die Methode *startFujaba* implementiert die Funktionalität des *Startknopfes*, sowie die Methoden *stopFujaba* bzw. *importFromFujaba* entsprechend die Funktionalitäten des *Stopknopfes* bzw. *Importknopfes* implementieren.

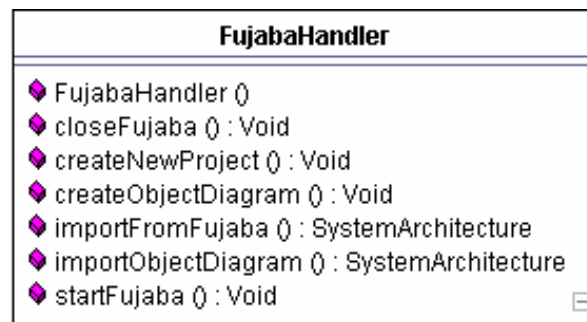


Abbildung 5.8.: FujabaHandler

- **FujabaHandler()**
Die Methode *FujabaHandler()* dient als Konstruktor für die *FujabaHandler*-Klasse.
- **CreateNewProjekt()**
Die Methode *CreateNewProjekt()* wird aus der Methode *startFujaba()* aufgerufen, sobald der *Fujaba*-Editor gestartet wird und legt ein neues *Fujaba*-Projekt an.
- **CreateOjektDiagramm()**
Diese Methode wird automatisch durch *CreateNewProjekt()* aufgerufen. Es wird ein „leeres“ Komponentendiagramm mit einem fest definierten Namen angelegt. Der Name der Methode „CreateOjektDiagramm()“ mag für die Erzeugung eines Komponentendiagramms falsch gewählt klingen, allerdings handelt es sich bei einem instanziierten Komponentendiagramm aus Sicht der Implementierung ebenfalls um ein Objektdiagramm.

- **ImportObjectDiagram()**

Die Methode *importObjectDiagram()* wird aus der Methode *importFromFujaba()* aufgerufen. Hierdurch wird der Parseralgorithmus gestartet und als Rückgabewert eine Systemarchitektur geliefert, die dann durch den FujabaHandler an den MainFrame des IFS-Editors weitergegeben wird.

5.3.2. Parseralgorithmus

Um das IFS-Modell zu übersetzen wurde ein Übersetzer entwickelt, der eine IFS-Modell-Instanz analysiert und dabei unmittelbar in ein IFS-Objekt übersetzt. Die lexikalische Analyse, die typischerweise die erste Stufe eines Compilers darstellt, wurde in die *set()*-Methoden der zu erzeugenden IFS-Objekte integriert [Alf02]. Fehlerhafte Werte werden soweit als möglich automatisch korrigiert, ansonsten zurückgewiesen. Der Parser ist fehlertolerant und ignoriert falsche Eingaben, wie z.B. unbekannte Objekte innerhalb des IFS-Modells. Abbildung 5.9 veranschaulicht die Objekthierarchie der Fujaba Datenstruktur des IFS-Modells. Der Parseralgorithmus übersetzt alle Objekte des IFS-Modells durch einen rekursiven Abstieg in der Datenstruktur (blaue und rote Pfeile) [Rei98, Alf99]. Die eigentliche Übersetzung des IFS-Modells ist Patternbasiert. So wurden für bestimmte Objekte der IFS-Modell-Instanz äquivalente Pattern in der IFS-Datenstruktur definiert. Findet der Parser nun ein ihm bekanntes Objekt, so wird automatisch eine Instanz des Patterns mit den Attributen des gefundenen Objektes angelegt. Einige Pattern können allerdings nicht vollständig in einem Schritt erzeugt werden, da dem Parser zum gegebenen Zeitpunkt noch Informationen fehlen. Diese Pattern werden dann zwischengespeichert und weiter verarbeitet, wenn der Parser die korrekte Bearbeitungsstelle erreicht hat. Die Transformation des Parseralgorithmus wurde durch die folgenden drei Klassen realisiert.

Synch2Fujaba

Die Synch2Fujaba-Klasse enthält die Methoden für die Transformation der IFS-Modell-Instanzen in die IFS-Objekte (vgl. Abbildung 5.10). Durch eine strukturierte Folge von Methodenaufrufen entsteht ein rekursiver Abstieg in der Objekthierarchie der Fujaba-Datenstruktur des IFS-Modells (vgl. Abbildung 5.9 blaue und rote Pfeile). Dabei werden bekannte Pattern (Komponenten mit ihren Ports, Objekte oder Teile der Zustandsautomaten) zu IFS-Objekten übersetzt. Diese Objekte werden in die Zieldatenstruktur eingebaut. Falls zu einem späteren Zeitpunkt noch einmal ein Zugriff auf diese Objekte notwendig ist, werden die Objektreferenzen zwischengespeichert. Die Speicherung der Objektreferenzen erfolgt mittels *Hashtables*, bei denen ein UML-Objekt als Schlüssel benutzt wird, um später noch einmal auf bereits angelegte Objekte zuzugreifen. Die Abbildung 5.10 veranschaulicht die verwendeten *Hashtables*. Dabei existieren Hashtables für die folgenden Objekte: *interface*, *protokoll*, *protokollMap*, *tpdClock* und *state* sowie für die Reference Signals: *globalDate*, *timeEvent*, *timerOrDeadline* und *referenceClock*.

Die Methoden der Klasse Synch2Fujaba implementieren das Interface Synch2IFS. Zu jeder *synch()* Methode der Strukturbeschreibung wird die Referenz auf das vorgefundene UML-Objekt als Parameter übergeben, wie im Klassendiagramm in Abbildung 5.10 zu sehen ist. Die oberste Ebene der Aufrufhierarchie ist die Methode *synch2System()*. Nach dem rekursiven Abstieg übergibt diese Methode eine Referenz auf das resultierende IFS-Objekt "*SystemArchitecture*" an den FujabaHandler. Im Folgenden wird der rekursive Abstieg ausführlich erläutert. Abbildung 5.9 visualisiert hierbei die wesentlichen Schritte des Parseralgorithmus.

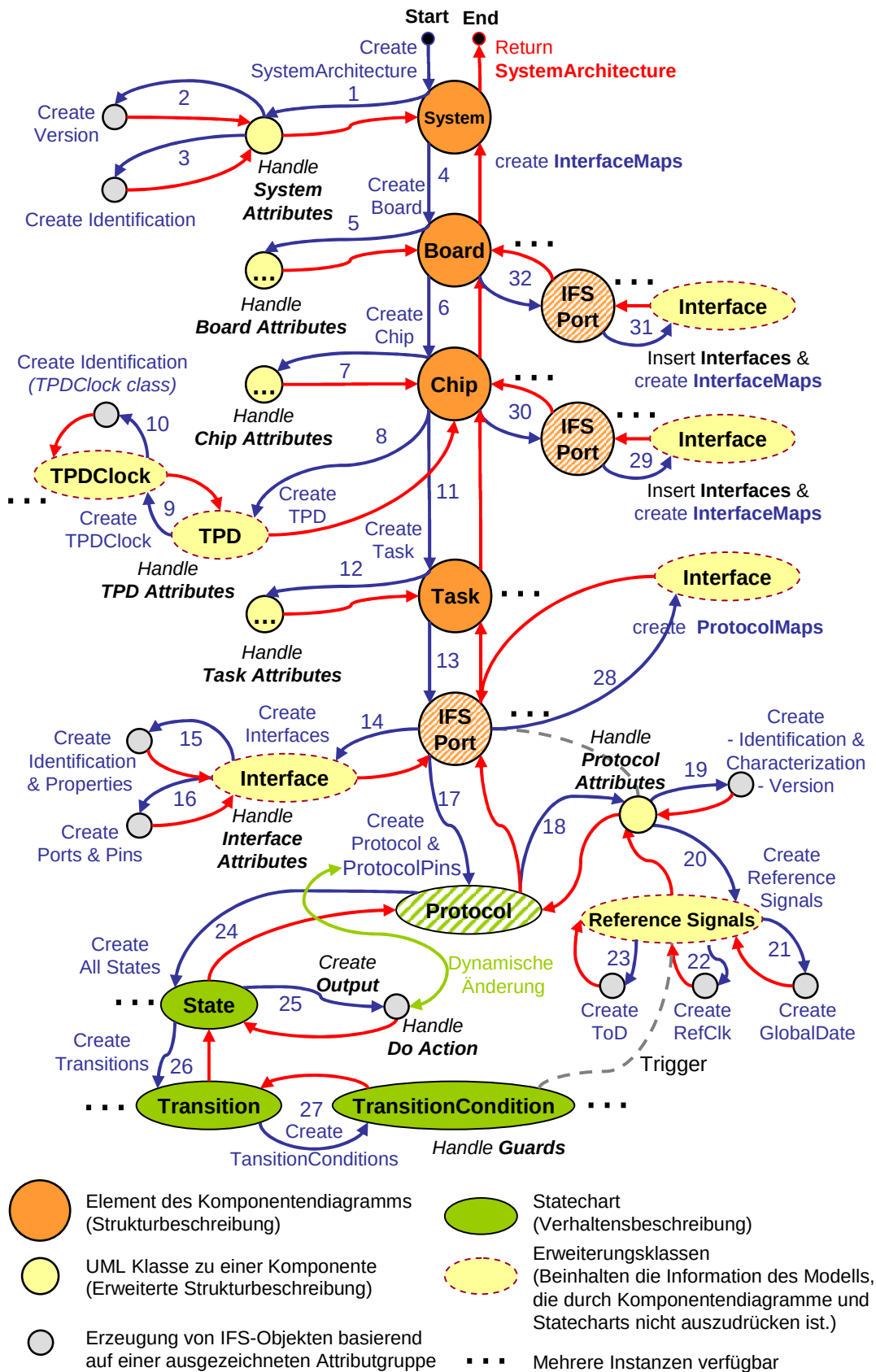


Abbildung 5.9.: Rekursiver Abstieg des Parseralgorithmus

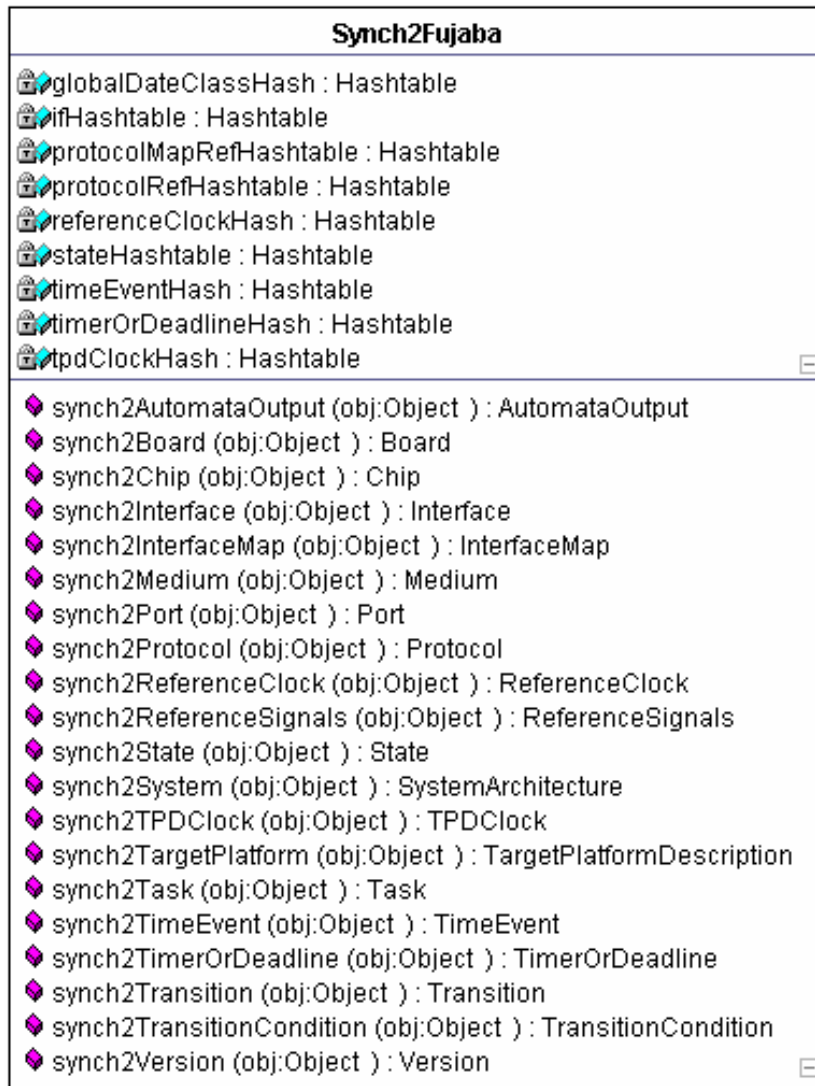


Abbildung 5.10.: Die Klasse “Synch2Fujaba”

System (*Synch2System()*)

Die *importObjectDiagram()* Methode des *FujabaHandlers* ruft als erstes die *Synch2System()* Methode der *Synch2Fujaba*-Klasse mit der Systemkomponente (UML-Objekt) des IFS-Modells als Parameter auf. Wie in Abbildung 5.9 dargestellt ist, repräsentiert der Knoten *System* den Einstiegspunkt des Parseralgorithmus. Während des Abstiegs werden sowohl vor (blaue Kanten), als auch nach dem Aufruf (rote Kanten) der Unterkomponenten, Operationen auf der aktuellen Ebene vorgenommen. Innerhalb der *Synch2System()* Methode wird aus dem übergebenen UML-Objekt zuerst das IFS-Objekt “*SystemArchitecture*” erzeugt. Als nächstes werden die Attribute des Systems ausgewertet. Dazu wird zuerst die Instanz einer festgelegten Klasse behandelt, die an eine jede Komponente gebunden ist und die Attribute der Identifikation eines IFS-Objektes beinhaltet. Es handelt sich hier um die Identifikation-Klasse des IFSKom-

5. Konzept und Implementierung der Modelltransformation

ponenten. Danach wird die Version durch die Methode `synch2Version()` erzeugt, die ein IFS-Objekt “*Version*” liefert. Dazu existiert auf der Systemebene die Klasse `Version`. Nachdem diese Parameter bearbeitet wurden, werden die Komponenten (UML-Objekte) der darunter liegenden Systemarchitekturhierarchieebene gesucht. Auf der Systemebene handelt es sich dabei um die Komponenten `Board` und `Medium`. Aus Gründen der Übersichtlichkeit wurden die Medien nicht in Abbildung 5.9 aufgenommen. Das Verhalten eines Mediums ist identisch zu dem einer Task, mit dem Unterschied, dass Medien auf allen Hierarchieebenen der Systemarchitektur vorkommen können. Die Referenzen der Unterkomponenten werden dann beim Aufruf der `synch2Board()` bzw. `synch2Medium()` Methode übergeben. Das hier im System vorgestellte Vorgehen wird auf gleiche Weise in den Komponenten `Board`, `Chip`, `Task` und `Medium` wiederholt (siehe Abbildung 5.9 Kanten 5, 7 und 12). Der Quellcode der `Synch2System()` Methode befindet sich im Anhang unter A.3.1.

Objekte werden im UML-Modell durch ihren Namen eindeutig gekennzeichnet. Die in IFS-Schema notwendigen IDs wurden daher nicht modelliert. Aus diesem Grund muss der Parser während der Erzeugung eines IFS-Objekts eine eindeutige ID hinzufügen. Der Parser benutzt dazu eine bereits vorhandene Methode der IFS-Datenstruktur.

Neben der Vergabe von IDs sorgt der Parseralgorithmus dafür, dass bei der Rückkehr aus dem rekursiven Abstieg, die *InterfaceMaps* angelegt werden. Dies geschieht überall dort, wo im UML-Modell zwei Interfaces durch Connectoren miteinander verbunden wurden (siehe Abbildung 5.9 Kanten 28, 29 und 31). Die *InterfaceMap* einer Komponente beinhaltet alle Verbindungen der Komponente selbst zu ihren Unterkomponenten, sowie alle Verbindungen ihrer direkten Unterkomponenten untereinander. Die einzige Ausnahme stellt das System dar, das selbst nicht verbunden sein kann, da es über keine Schnittstellen verfügt. *InterfaceMaps* werden nur dann angelegt, wenn die entsprechenden Interfaces vorhanden sind. Daher erzeugen die `synch2...` Methoden die erforderlichen IFS-Objekte “Interface” bereits, bevor die *InterfaceMaps* behandelt werden.

Boards (`synch2Board()`)

Die `synch2Board()` Methode funktioniert in ähnlicher Weise wie die `Synch2System()` Methode. Dort wird aus dem übergebenen UML-Objekt zuerst das IFS-Objekt “*Board*” mit den entsprechenden Attributen erzeugt. Danach wird die Methode `synch2Version()` aufgerufen, um das benötigte IFS-Objekt “*Version*” zu erzeugen. Als nächstes werden die vorhandenen Unterkomponenten (UML-Objekte) `Chip` und `Medium` gesucht, und die Methode `synch2Chip()` bzw. `synch2Medium()` aufgerufen.

Chip (`synch2Chip()`)

Ebenso wie bei der `Synch2System()` und `synch2Board()` Methode, legt die `synch2Chip()` Methode ein IFS-Objekt “*Chip*” mit den dazu gehörenden Attributen und dem IFS-Objekt “*Version*” an. Dann wird die Instanz der TPD-Klasse (UML-Objekt) auf der Chipebene gesucht, und ihre Referenz dem `synch2TPD()` Methodenaufruf übergeben. Als nächster Schritt werden die Task- und Mediumkomponenten auf Chipebene gesucht, die Referenzen werden einzeln bei dem `synch2Task()` bzw. dem `synch2Medium()` Methodenaufruf übergeben.

TPD (*synch2TPD()*)

Innerhalb der *synch2TPD()* Methode wird aus dem übergebenen UML-Objekt das IFS-Objekt “*TPD*” mit entsprechenden Attributen erzeugt (siehe Abbildung 5.9 Kanten 8). Die Attribute des *TPD* werden aus der Instanz der *TPD*-Klasse abgeleitet. Als nächstes werden auf der Chi-pebene alle vorhandenen Instanzen der *TPDClock*-Klasse gesucht. Deren Referenz werden bei dem Methodenaufruf *synch2TPDClock()* übergeben.

TPDClock (*synch2TPDClock()*)

Wie bereits in Kapitel 4.2.2 erwähnt wurde, werden alle *TPDClocks* als Instanz der *TPDClock*-Klasse auf der Chi-pebene modelliert. Die *synch2TPDClock()* Methode wird dann anhand dieser Instanzen das entsprechende IFS-Objekt “*TPDClock*” einzeln erzeugen (siehe Abbildung 5.9 Kanten 9 und 10).

Task (*synch2Task()*)

Synch2Task() ist eine komplexe Methode, die zusätzlich bei der Erzeugung des IFS-Objektes “*Task*” neben den zugehörigen Attributen und der Version, die Protokollaspekte des modellierten Zustandautomaten berücksichtigt. Für die Erzeugung des Protokolls werden zuerst alle Ports der Task (UML-Objekte) gesucht und einzeln wie folgt abgearbeitet. Die Referenzen eines Ports auf seine *ProvidedInterfaces* bzw. *RequiredInterfaces* (UML-Objekt) werden ermittelt und dann dem Methodenaufruf *synch2Interface()* übergeben. Dabei ist zu beachten, dass im IFS-Modell genau ein Interface pro Port erlaubt ist (siehe Kapitel 4.2.1).

- **Interfaces (*synch2Interface()*)**

Innerhalb der *synch2Interface()* Methode wird anhand der modellierten Instanz der Interfaceklasse ein IFS-Objekt “*Interface*” mit den entsprechenden Identification- und Properties-Attributen erzeugt (siehe Abbildung 5.9 Kanten 14 und 15). Dabei werden die Properties-Attribute durch die Methode *synch2Properties()* erzeugt. Diese Instanz wurde bei der Modellierung der IFS-Modell-Instanz in der Systemebene abgelegt (siehe Kapitel 4.2.1). Danach wird die *synch2Port()* Methode mit der gleichen Interfacereferenz aufgerufen.

- **Port (*synch2Port()*)**

Alle erforderlichen Signale für die Verhaltensbeschreibung des Protokolls wurden bei der Modellierung der IFS-Modell-Instanz in der Instanz der Interfaceklasse deklariert (siehe Kapitel 4.2.1). Die *synch2Port()* Methode erzeugt aus allen deklarierten Signalen zuerst das IFS-Objekt “*Port*” (siehe Abbildung 5.9 Kante 16). Die Identifikationsattribute werden anhand des Signalnamens automatisch generiert. Danach wird die Anzahl der Pins des Ports anhand des Initialwerts des Signals ermittelt. Als Beispiel wird aus der Deklaration: Signal X_Motor = “000”, der PortName X_Motor zugewiesen und eine Pinanzahl von drei festgestellt. Als nächstes wird die *synch2pin()* Methode für alle Pins aufgerufen bis alle IFS-Objekte “*Pin*” erzeugt wurden. Dadurch entsteht die *PinList* des Ports.

- **Pin (*synch2Pin()*)**

Die *synch2Pin()* Methode legt ein IFS-Objekt “*Pin*” an (siehe Abbildung 5.9 Kante 16). Die Identifikationsattribute werden automatisch aus dem Portnamen generiert. Bezüglich der Characterization-Attribute wird die Signalrichtung *Direction* hier initial auf den Wert “*Other*” und der Type auf “*Std_Logic*” gesetzt.

5. Konzept und Implementierung der Modelltransformation

An dieser Stelle ist bereits das IFS-Objekt “Interface” eines Task-Ports (UML-Objekt) mit allen zugehörigen Attributen und der PortList angelegt. Die Referenzen der “Interfaces” werden in der ifHashtable zwischengespeichert und werden später für die Erzeugung des IFS-Objekts “Protocol” und der Interfaces in den höheren Hierarchieebenen benutzt. Als nächstes wird die `synch2Protocol()` Methode mit der Referenz auf den Port der Task aufgerufen.

- **Protocol (`synch2Protocol()`)**

Wie bei allen `synch()` Methodeaufrufen wird auch hier ein IFS-Objekt (“Protocol”) angelegt. Im Gegensatz zu den anderen `synch()` Methoden wird das Protocol durch die Methode `generateProtocolFromInterface()` der InterfaceProtocolFactory-Klasse erzeugt. Bei dem Aufruf der `generateProtocolFromInterface()` Methode wird die Referenz des bereits erzeugten Interfaces übergeben. Aus den vorhandenen Port-Interfaces wird das IFS-Objekt “ProtokollPins” generiert (siehe Abbildung 5.11).

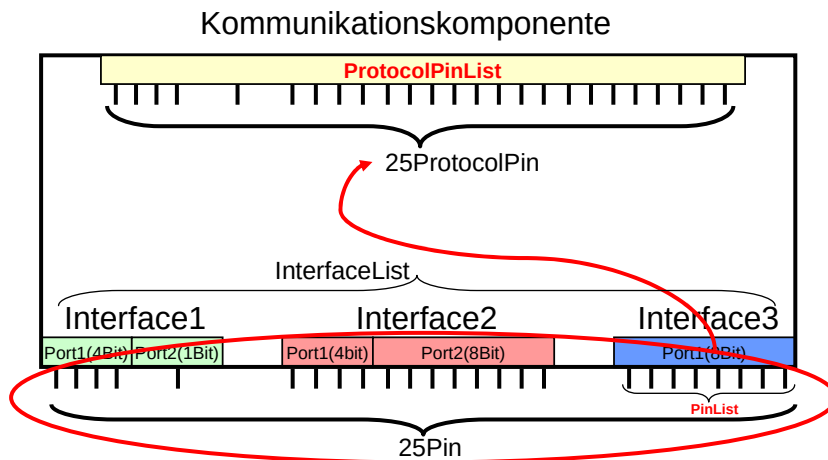


Abbildung 5.11.: Generierung der ProtocolPins

Die ProtokollPinListen werden in einer Hashtable zwischengespeichert, bei der der Name des zugehörigen IFS-Objekts “Port” als Schlüssel benutzt wird. Die gespeicherten ProtocolPins werden später für die TransitionCondition genutzt. Die InterfaceProtocolFactory-Klasse wird in einem eigenen Unterpunkt mit ihrer Methode beschrieben. Der Quellcode befindet sich im Anhang unter A.3.2 Die Identification sowie die Characterization-Attribute und ReferenceSignals des Protocols werden bei der Modellierung der Portinstanz deklariert (siehe Abbildung 4.13). Aus dieser Deklaration werden zuerst die “Protocol” Attribute ermittelt, dann wird ein IFS-Objekt “Version” aus der Task-Version erzeugt. Danach wird die GlobalDate-, TimerOrDeadline- und ReferenceClock-Klasse aus der Typedeklaration innerhalb der Portklasse ermittelt. Danach wird überprüft, ob das dazu gehörende IFS-Objekt schon in der entsprechenden Hashtable ist, bevor `synch2GlobalDate()` bzw. `synch2TimerOrDeadline()` oder `synch2ReferenceClock()` aufgerufen wird. Die legt die entsprechenden IFS-Objekte (siehe Abbildung 5.9 Kanten 20, 21, 22, 23) an und speichert die Referenz in der Hashtable. Die Methode `synch2ReferenceSignals()` wurde nicht benutzt. Ein IFS-Objekt “ReferenceSignal” wird bei Bedarf (Vorhanden GlobalDate bzw. TimerOrDeadline oder ReferenceClock) direkt in der Protokollebene erzeugt.

Nachdem das IFS-Objekt “*Protocol*” mit dem zugehörigen ProtocolPins erzeugt wurde, wird als nächstes die *ProtocolStateMachine* des Ports betrachtet, um die weiteren Protokollaspekte wie Zustände und Änderungen der Charaterization Attribute der ProtocolPins zu berücksichtigen. Die Zustandstransitionen können erst dann angelegt werden, wenn alle IFS-Objekte “*State*” vorhanden sind. Daher wird als erstes zu jedem der Zustände der *ProtocolStateMachines* die *synch2State()* Methode aufgerufen (siehe Abbildung 5.9 Kanten 24).

- **State (*synch2State()*)**

Innerhalb der *synch2State()* Methode wird ein IFS-Objekt “*State*” mit dem Identifikationsparameter erzeugt. Danach werden alle AutomataOutputs (Automatenausgaben) aus dem UML-Object “*DoAction*” mit der zugehörigen ProtocolPinID ermittelt. Als nächstes wird die *synch2AutomataOutput()* Methode aufgerufen. Der “*State*” wird in der state-Hashtable zwischengespeichert.

- **AutomataOutput (*synch2AutomataOutput()*)**

Die *DoAction* weist den Signalen, die in der Portinterfaceklasse deklariert sind, gemäß der in Kapitel 4.2.2 (vgl. Abbildung 5.2) beschriebenen Syntax für bestimmte Zustände Werte zu. Anhand der angegebenen Werte wird ein IFS-Objekt “*AutomataOutput*” mit entsprechenden Attributen erzeugt. Die Charaterization-Attribute des Protokollpins werden an dieser Stelle angepasst (siehe Abbildung 5.9 Kante 25).

Alle Protokollzustände wurden bereits erzeugt und in der stateHashtable zwischengespeichert. Für jeden dieser Zustände der ProtocolStateMachine (Fujaba-Objekt) werden nun alle Transitionen gesucht und mit der *synch2Transition()* Methode einzeln bearbeitet.

- **Transition (*synch2Transition()*)**

Ein IFS-Objekt “*Transition*” wird innerhalb des *synch2Transition()* Methode erzeugt. Die Identifikationsattribute werden aus der *UMLRealtimeTransition* (Fujaba-Objekt) generiert. Die *NextStateID* wird mit Hilfe der stateHashtable ermittelt. Anschließend werden alle IFS-Objekte “*TransitionCondition*” der bereits erzeugten “*Transition*” angelegt. Dafür wird die *synch2TransitionCondition()* Methode mit dem *Guard-Informationen* aufgerufen.

- **TransitionCondition (*synch2TransitionCondition()*)**

Die *synch2TransitionCondition()* Methode legt ein IFS-Objekt “*TransitionCondition*” an. Der *Transitionguard* enthält Informationen mit einer festgelegten Syntax (siehe Kapitel 4.2.2 (vgl. Abbildung 5.2)). Anhand dieser Informationen werden die TransitionCondition-Attribute sowie die TransitionType-Attribute der “*Transition*” ermittelt.

InterfaceProtocolFactory

Die Klasse InterfaceProtocolFactory (siehe Abbildung 5.12) enthält zwei wesentliche Methoden. Zum einen die Methode *generateProtocolFromInterface()*, welche als Argument eine IFS-Objekt “*Interface*” erwartet, und dann ein IFS-Objekt “*Protocol*” mit ProtocolPins erzeugt. Die ProtocolPins werden aus der Interface-PortList generiert (siehe Java Code). Zum anderen erwartet die Methode *generateInterfaceFromProtocol()* als Argument ein IFS-Objekt “*Protocol*” und legt ein IFS-Objekt “*Interface*” mit den zugehörigen Ports an. Die Ports werden aus den ProtocolPins ermittelt. Die beiden Methoden speichern zu jeder PortList eine Hashtable mit dem portName als Schlüssel.

5. Konzept und Implementierung der Modelltransformation

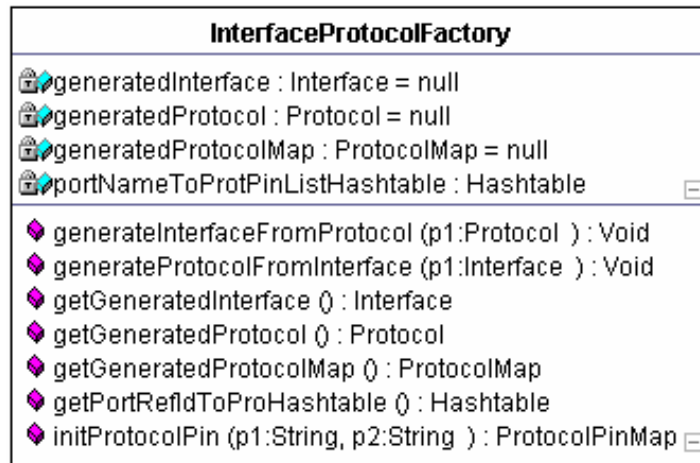


Abbildung 5.12.: Die Klasse “InterfaceProtocolFactory”

```

/**
 * Generates a protocol and a protocol map from an interface.
 * @param inter Interface
 */
public void generateProtocolFromInterface(Interface inter)
{
    String newDefaultID = inter.getID();

    generatedProtocol = new Protocol(newDefaultID);
    generatedProtocol.setName(inter.getName() + "_GenPr");
    generatedProtocol.setDescription("Generated_Protocol_from_Interface_"
        + inter.getName() + "_(" + inter.getID() + ")");

    generatedProtocolMap = new ProtocolMap();
    generatedProtocolMap.setID(newDefaultID);
    generatedProtocolMap.setName("I" + inter.getID() + "_P_GenPrMap");
    generatedProtocolMap.setDescription("Generated_Map_from_Interface_"
        + inter.getName() + "_(" + inter.getID() + ")");
    generatedProtocolMap.setInterfaceID(inter.getID());
    generatedProtocolMap.setProtocolID(generatedProtocol.getID());

    ProtocolPin protocolPin;
    Port port;
    Pin pin;
    for (int i = 0; i < inter.getPortListSize(); i++)
    {
        port = inter.getPort(i);
        X4JVector protPinList = new X4JVector();
        for (int j = 0; j < port.getPinListSize(); j++)
  
```

```

{
    // create ProtocolPin from Pin
    pin = port.getPin(j);
    protocolPin = new ProtocolPin();
    protocolPin.setName(pin.getName());
    protocolPin.setID(
        this.generatedProtocol.getProtocolPinList().getFreeID());
    protocolPin.setDescription(pin.getDescription());
    protocolPin.setDirection(pin.getDirection());
    protocolPin.setType(pin.getType());

    // add ProtocolPin to Protocol
    this.generatedProtocol.addProtocolPin(protocolPin);

    // add ProtocolPin.clone in Vector
    protPinList.add(protocolPin);
    // create ProtocolPinMap & add to ProtocolMap
    generatedProtocolMap.addProtocolPinMap(initProtocolPin("Port",
        port.getID(), pin.getID(), protocolPin.getID()));
}
// Put vector to PinList in to Hasch
portNameToProtPinListHashtable.put(port.getName(), protPinList);
}
} // generateProtocolFromInterface

```

Zusätzlich enthält die Klasse die Methode *getGeneratedInterface()*, *getGeneratedProtocol()* und *getGeneratedProtocolMap()* für die Rückgabe des erzeugten Interfaces (*generatedInterface*) bzw. Protocols (*generatedProtocol*) oder ProtocolMap (*generatedProtocolMap*), sowie die Methode *getPortRefidToProHashtable()* für die Rückgabe der ProtocolPinHashtable (*portNameToProtPinListHashtable*). Die letzte Methode *initProtocolPin()* erwartet vier Strings als Argument und erzeugt daraus ein IFS-Objekt ProtocolMap (Siehe Java Code). Die ProtocolMap bildet die ProtocolPins auf die physikalischen Pins ab.

```

/**
 * Returns a configured Protocol Map.
 * @return ProtocolMap
 */
public ProtocolPinMap initProtocolPin(String category,
    String portID, String pinID, String protocolPinID)
{
    ProtocolPinMap protocolPinMap = new ProtocolPinMap();
    protocolPinMap.setInterfaceCategory(category);
    protocolPinMap.setInterfacePortID(portID);
    protocolPinMap.setInterfacePinID(pinID);
    protocolPinMap.setProtocolPinID(protocolPinID);
    return protocolPinMap;
} // getGeneratedInterface

```

FujabaTools

Die Klasse `FujabaTools` (siehe Abbildung 5.13) ist eine sehr wichtige Klasse für den Parseralgorithmus. Die Methoden dieser Klasse parsen die als Strings übergebenen Ausdrücke gemäß der IFS-Modell-Syntax und geben bestimmte Stringwerte zurück. Fehlerhafte Werte werden soweit als möglich automatisch korrigiert, ansonsten zurückgewiesen.

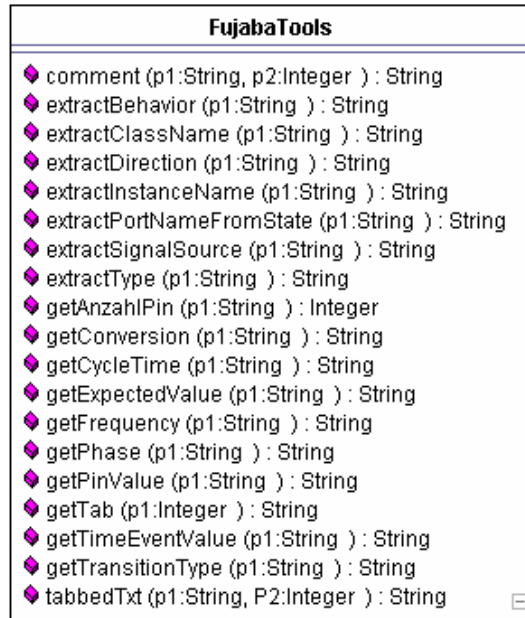


Abbildung 5.13.: Die Klasse “FujabaTools”

Als Beispiel kann die *DoAction* eines Zustandsautomaten folgende Werte, gemäß der in Kapitel 4.2.2 (vgl. Abbildung 5.2) beschriebenen Syntax enthalten: `C.I:X_Motor = “00101”`; `C.O:Y_Motor = “001”`. Dieser String wird bei dem Semikolon “;” geteilt. Aus dem entstehenden String, z.B. `C.I:X_Motor = “00101”` kann die notwendige Information mit dem entsprechenden Methodenaufruf extrahiert werden. Ein String wird bei dem Methodenaufruf als Parameter übergeben, in diesem Fall der String: `C.I:X_Motor = “00101”`.

extractPortNameFromState()

Die Methode *extractPortNameFromState()* liefert den PortName als Rückgabewert zurück. In diesem Fall wird “*X_Motor*” als String zurückgeliefert.

extractDirection()

Hier wird die Direction extrahiert. Für dieses Beispiel wird “Input” als Rückgabewert zurückgegeben.

extractBehavior()

Das Behavior wird bei diesem Methodenaufruf zurückgeliefert. In diesen Fall wird der Sting “Control” übergeben. Die *extractBehavior()* Methode sieht wie folgt aus.

```

/*
 * Methode um aus "C.I:X_Motor = '00101'" das Behavior C(Control) zu extrahieren
 * Do action liefert
 * zB. C.I:X_Motor = '00101'; C.O:Y_Motor = '001' dann split auf ";"
 */
public static String extractBehavior(String string)
{
    if (string.length() > 0)
    {
        int index = -1;
        index = string.indexOf(".");
        if (index >= 0)
        {
            String value = (string.substring(0, index)).replaceAll("_", "");
            if (value.equalsIgnoreCase("D"))
                return "Data";
            else if (value.equalsIgnoreCase("C"))
                return "Control";
        }
    }
    return null;
} // extractBehavior

```

getPinValue()

Bei diesem Methodenaufruf wird ein Stringwert und ein Integerwert übergeben, Z.B. getPinValue ("C.I:X_Motor = "00101"", 3). Der Signalwert der angegebenen Stellen wird als String zurückgegeben. In diese Fall "1".

Gemäß der IFS-Modell-Syntax werden die *Guard-Informationen* der Transitionen ebenso wie bei der *DoAction* des Zustandsautomaten mit vier Methoden extrahiert. Das sind die Methoden *getTransitionType()*, *getExpectedValue()*, *extractSignalSource()*, *extractPortNameFromState()*.

Die Attribute des IFS-Objekts "*TransitionCondition*" sowie die Characterization-Attribute der "*ProtocolPincolPins*" werden anhand dieser Informationen ausgewertet.

extractClassName(), extractInstanceName()

Die *extractClassName()* und *extractInstanceName()* Methode erwarten ein String als Argument. Aus dem String wird der Klassenname bzw. Komponentennamen sowie der Name ihrer Instanz ermittelt. Dazu wird zuerst die Instanz einer festgelegten Klasse behandelt, die an eine jede Komponente gebunden ist.

Bei der Modellierung der IFS-Modell-Instanz werden Werte für die Attribute der Instanzklasse zugewiesen. Diese Werte werden für die Erzeugung des entsprechenden IFS-Objekts verwendet. Allerdings ist für die Attributwerte bestimmter Kategoriegruppen eine Konsistenzprüfung erforderlich. Die *getPhase()* sowie *getFrequency()* und *getCycleTime()* Methoden prüfen die *Phase*-, *CycleTime*- bzw. *Frequenz-Attribute* und weisen die entsprechenden Werte gemäß der Zielsprachensyntax gegebenenfalls zurück.

5. Konzept und Implementierung der Modelltransformation

getTimeEventValue()

Für die Zuweisung von Attributen an TimeEventValue werden Abkürzungen verwendet. Aus dem übergebenen String wird dann der eigentliche Werte des TimeEventValues generiert (siehe Quellcode).

```
/*
 * return "High Peak" für den String "HP" bzw. "H" ...
 */
public static String getTimeEventValue(String string)
{
    if (string.equals("1"))
        return "Logic_One";
    else if (string.equals("0"))
        return "Logic_Zero";
    else if (string.startsWith("H") || string.startsWith("h"))
        return "High_Peak";
    else if (string.startsWith("L") || string.startsWith("l"))
        return "Low_Peak";
    else if (string.startsWith("S") || string.startsWith("s"))
        return "Signal_Changed";
    else
        return null;
} // getTimeEventValue
```

extractType()

Anhand der Signaldeklaration wird innerhalb der *extractType()* Methode der Typ des Signals aus dem angegebenen String identifiziert. Der Quellcode befindet sich im Anhang unter A.3.3.

Comment(), tabbedTxt() und getTab()

Diese Methoden sind da, um Debug-Informationen strukturiert und übersichtlich auszugeben.

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

Bisher basierte die Modellierungssprache der Interface Synthese (IFS-Format) auf einem XML Schema. Im Rahmen dieser Arbeit wurde ein Modellierungskonzept in Form eines UML2.0 Profils für das IFS-Format definiert, um eine standardisierte und intuitive Nutzung der Interface Synthese zu ermöglichen. Unter dem Begriff Modelltransformation wurden verschiedene Möglichkeiten für die Anbindung des UML2.0 Modells an das bestehende Modellierungskonzept vorgestellt. Anschließend folgte eine Erläuterung des implementierten Transformationsalgorithmus.

Kapitel zwei beschreibt die Grundlagen, auf denen diese Arbeit aufbaut. Dabei wurden die für die Modellierung in UML benötigten Diagrammtypen eingeführt sowie wichtige Aspekte und Eigenschaften näher erläutert. Darauf folgte eine Einführung in das IFS-Format, um das grundlegende Verständnis für die in Kapitel drei und vier präsentierte UML2.0 Modellierung zu schaffen.

Das in dieser Arbeit in Kapitel vier definierte UML2.0 Profil wurde als IFS-Modell bezeichnet. Basierend auf dem vordefinierten IFS-Format verfügt das IFS-Modell über eine spezielle Semantik sowie gewisse Syntaxeinschränkungen. Dabei wurde das Stereotypenkonzept für die Erweiterung des UML2.0 Metamodells eingesetzt. Das resultierende IFS-Modell dient von nun an als Metamodell für die Modellierung von IFS-Modell-Instanzen.

Neben dem angewendeten Transformationskonzept wurden weitere mögliche Konzepte und Realisierungen der Modelltransformation in Kapitel drei vorgestellt und bewertet. Der im Rahmen dieser Arbeit implementierte Transformationsalgorithmus wurde in Kapitel fünf ausführlich erläutert. Der darin beschriebene Übersetzer beinhaltet einen Parser, der einen rekursiven Abstieg zur Erzeugung der Zieldatenstruktur durchführt. Die hierbei verwendeten Techniken basieren auf wohl bekannten Konzepten des Compilerbaus.

Ein detailliertes Beispiel für die Nutzung des definierten UML2.0 Profils findet sich im Anhang unter A.1. Das vorgestellte Modell ist mit Hilfe des Demonstrators (Fujaba und IFS-Editor) umgesetzt und validiert worden. Das Beispiel unterstreicht insbesondere die praktische Anwendbarkeit des Profils.

6.2. Ausblick

Der entwickelte Parseralgorithmus ist fehlertolerant und ignoriert falsche Eingaben wie z.B. unbekannte Objekte innerhalb des IFS-Modells. Fehlerhafte Werte werden soweit als möglich automatisch korrigiert, ansonsten zurückgewiesen, ohne eine Mitteilung an den Modellierer zu geben. In Zukunft könnte die Funktionalität des Parseralgorithmus erweitert werden, um zusätzlich zur Konsistenzprüfung von Eingabeparametern eine Mitteilung an den Modellierer zu liefern. Die Vernachlässigung von fehlerhaften Eingaben kann sonst zu Inkonsistenzen zwischen dem Modell und der generierten Systemarchitektur führen. Deswegen wäre eine Option wünschenswert, in der es eine Möglichkeit für den Designer gibt, interaktiv Veränderungen des bereits erstellten IFS-Modells durchzuführen.

Eine elegante Form der Werkzeugkopplung könnte anstelle der Import-Funktion durch die online Synchronisation zwischen den Modellen von Fujaba- und dem IFS-Editor erreicht werden. Beide Datenstrukturen müssten dann zur Laufzeit ständig miteinander abgeglichen werden, um die Konsistenz der Modellinstanzen zu gewährleisten.

Im aktuellen UML2.0 Profil wird nur die Kommunikation zwischen Komponenten mit gleichen Schnittstellen betrachtet. Dies beruht unter anderem darauf, dass in Fujaba nur Verbindungen (Connectors) zwischen kompatiblen Schnittstellen erzeugt werden können. Die Verbindung und Synthese von inkompatiblen Schnittstellen verbleibt eine Eigenschaft des IFS-Editors. In Zukunft wäre es denkbar, auch den Kommunikationswunsch von Komponenten mit inkompatiblen Schnittstellen unter der Benutzung einer IFB-Komponente (Interface Block) zu modellieren. Damit würde nur die eigentliche Synthese weiterhin im IFS-Editor stattfinden.

A. Anhang

A.1. Beschreibung eines Beispielsmodells

Das Modellierungskonzept sowie die verschiedenen Ansätze für die Modelltransformation sind bereits in Kapitel drei und vier unabhängig von einer konkreten Modellierungsumgebung vorgestellt worden. In diesem Kapitel wird das Modellierungskonzept anhand eines Beispiels erläutert. Das in Kapitel 4.2.1 vorgestellte Kommunikationssystem “*sys1*” (siehe Abbildung 4.5) wird hier mit Hilfe des Fujaba-Editors schrittweise modelliert. Dabei wird bei der Modellierung der IFS-Modell-Instanz “*sys1*” das in Kapitel 4 vorgestellte IFS-Profil intern auf die bestehende Fujaba Datenstruktur unter Benutzung der Grafiknotation abgebildet. Durch das Modellierungsbeispiel wird gleichzeitig der Umgang mit dem Fujaba-Editor vermittelt. Bei der Modellierung der IFS-Modell-Instanz “*sys1*” müssen folgende Schritte durchgeführt werden. Zuerst werden die IFS-Komponenten mit UML-Diagrammen entworfen. Dazu wird der *Fujaba-Editor* per Klick auf *Start Fujaba* des *IFS-Editors* gestartet.



Abbildung A.1.: IFS-Editor: Start Fujaba

A.1.1. Die grafische Oberfläche

Die Abbildung A.2 stellt die grafische Oberfläche (*GUI, graphical user interface*) des Fujaba-Editors dar:

A.1.2. Komponente und Objektdiagramm

Komponente

Bei der Modellierung der IFS-Modell-Instanz “*sys1*” wird zuerst die Komponente System erstellt. Nach einem Klick auf den Knopf Create a new Component (siehe Abbildung A.3) wird ein kleines Fenster angezeigt, in dem der Name der Komponente angegeben wird (siehe Abbildung A.4).

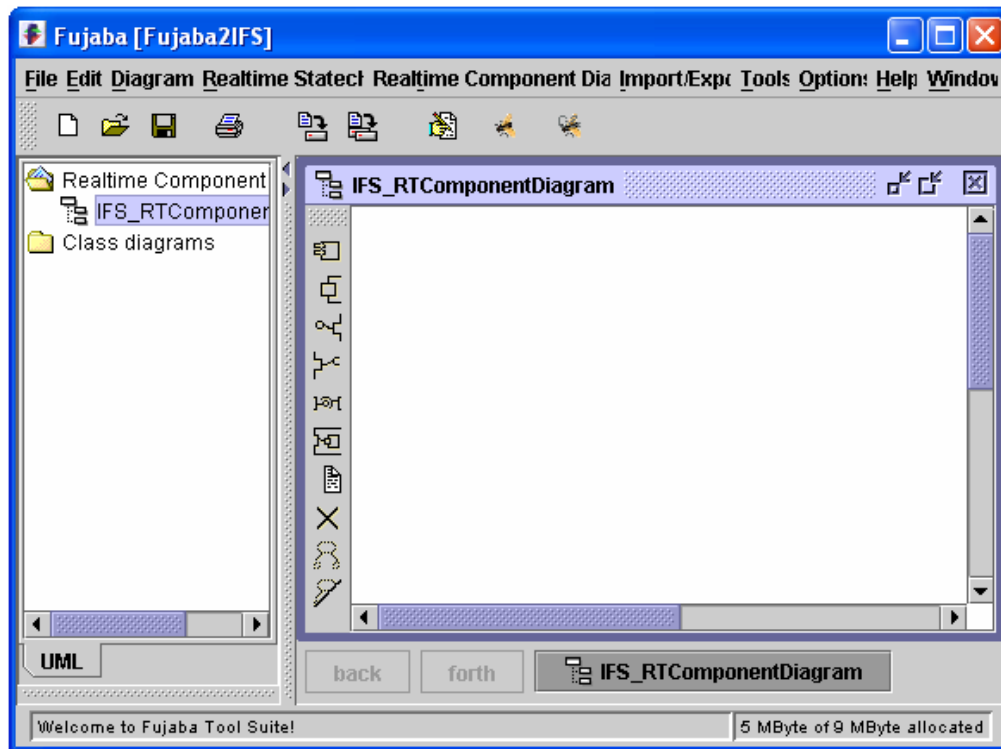


Abbildung A.2.: Fujaba Editor



Abbildung A.3.: Knopf "Create a new Component"

Die Komponenten unterscheiden sich nur in ihrer Bezeichnung; von daher ist es notwendig, zuerst einen eindeutigen Namen für die Komponente einzugeben und diesen mit dem Namen des Typen zu verbinden. In diesem Fall ergibt dies *sys1:System* (siehe Abbildung A.4). Der Doppelpunkt wird für die Trennung zwischen Namen und Typ verwendet.

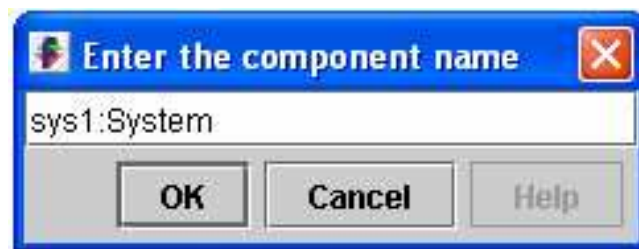


Abbildung A.4.: Maske zur Eingabe des Komponentennamens

Objektdiagramm

Nach der Generierung der Komponente *“System”*, werden automatisch die dazu gehörenden Objektdiagramme (siehe Abbildung A.5) als Referenz erzeugt.

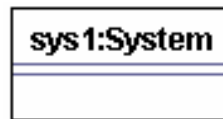


Abbildung A.5.: Das Objektdiagramm “sys1System”

Eingabe von Attributen

Die Objektdiagramme ermöglichen bei der Eingabe von bestimmten Attributen die Erzeugung gültiger Instanzen des IFS-Schemas für die entsprechende Komponente. Mit einem Linksklick auf das Objektdiagramm *“sys1:System”* wird ein Kontextmenü mit mehreren Einträgen geöffnet, die für die Modellierung des UML Objektdiagramms benötigt werden (siehe Abbildung A.6).

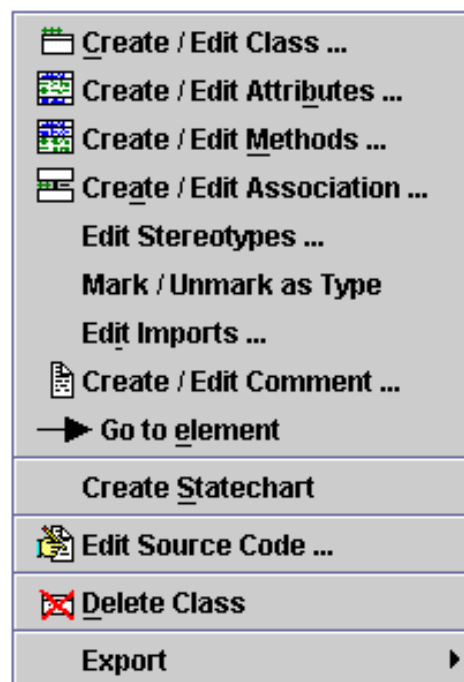


Abbildung A.6.: Kontextmenü für ein Klassen- bzw. Objektdiagramm

Der Eintrag *“Create / Edit Attributes”* ist für die Erzeugung bzw. Änderung der Attributwerte vorgesehen. Der Eintrag *“Create / Edit Comment”* kann für die Beschreibung (Description) der Komponente benutzt werden. Alle anderen Kontextmenüpunkte sind für diese Aufgabe nicht relevant. Nach Druck auf den Knopf *“Create / Edit Attributes”* wird das folgende Fenster

angezeigt. Dort wird der Name des Attributes (Attribute Name), der dazugehörige Wert (Initial Value) sowie der Typ (Types) eingegeben. Der Typ “String” wird in diesem Fall für alle Attribute verwendet.

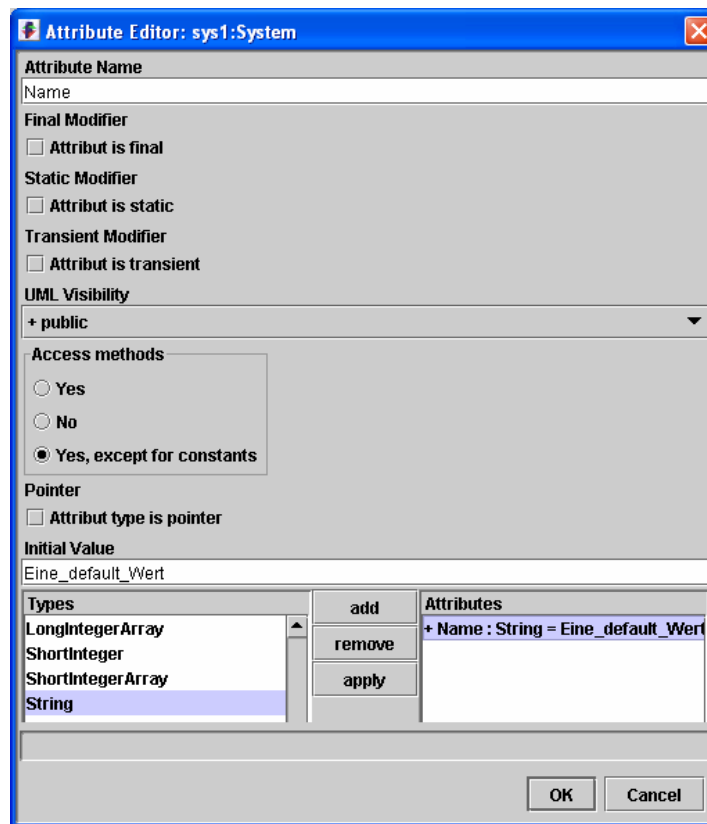


Abbildung A.7.: Eingabefenster für die Attribute

Die Attribute werden mit der Schaltfläche “Add” hinzugefügt. Nach Eingabe von den Attributen wird die Eingabe mit der Schaltfläche OK bestätigt. Die eingefügten Parameter werden in das Objektdiagramm übernommen (siehe Abbildung A.8). Es handelt sich hier um die Attribute der *Identification-Klasse* (siehe Kapitel 4).

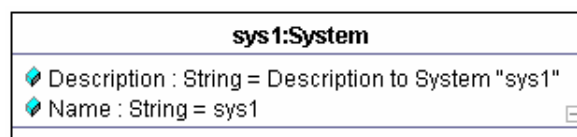


Abbildung A.8.: Attribute der Klasse “System”

Der *Fujaba-Editor* erzeugt eine eigene interne Identifizierung und Referenzierung für jedes modellierte Objekt. Die *Description* kann ebenso als Attribut oder als Kommentar eingegeben werden, je nachdem, ob es sich um eine kurze oder lange Eingabe handelt (siehe Abbildung A.9).

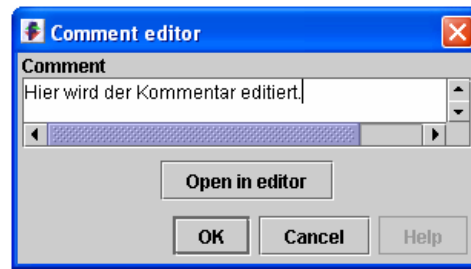


Abbildung A.9.: Kommentarfenster

Subkomponenten

Als nächstes werden die Subkomponenten modelliert. Mit einem Linksklick auf die Systemkomponente “*sys1*” wird ein kleines Auswahlfenster mit zwei Möglichkeiten angezeigt (siehe Abbildung A.20). Die erste Auswahl steht für die Erstellung von Subkomponenten zur Verfügung. Die zweite Auswahl wird an entsprechender Stelle erläutert.

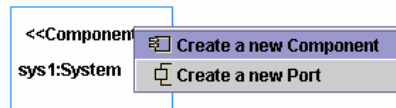


Abbildung A.10.: Erzeugung von Subkomponenten

Entsprechend der Modellierung der Komponente *sys1* werden alle Subkomponenten modelliert (siehe Abbildung A.11).

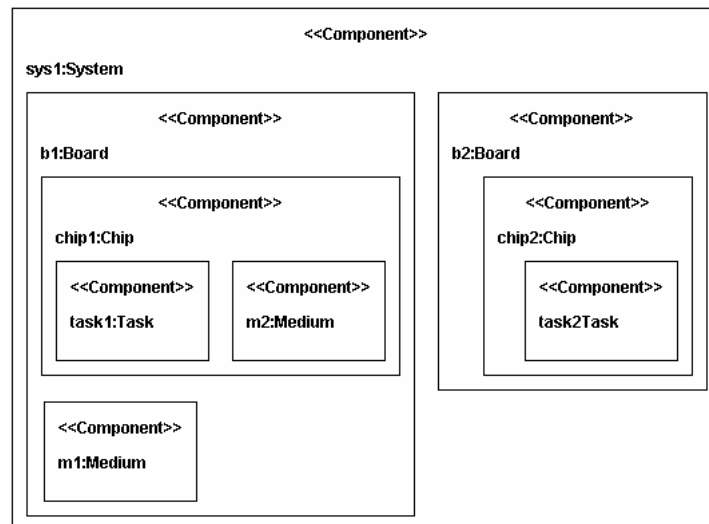


Abbildung A.11.: Komponenten der Systemarchitektur

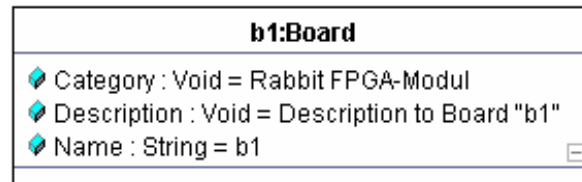


Abbildung A.12.: Instanz der Komponente Board

Danach werden die Identifikationsattribute in dem entsprechenden Objektdiagramm der jeweiligen Komponenten eingegeben, beispielsweise für die Komponente “Board” (siehe Abbildung A.12). Für die *Category* des Identifikationsattributes bietet das IFS-Format mittels der Technik der *Enumerations* mehrere Typen zur Auswahl an. Diese Typen sollen bei Eingabe des Attributs *Category* in zugehörige Objektdiagramme eingegeben werden. Folgende Abbildungen zeigen die expliziten Typen, die zurzeit vom IFS-Format unterstützt werden.

Board-Kategorie

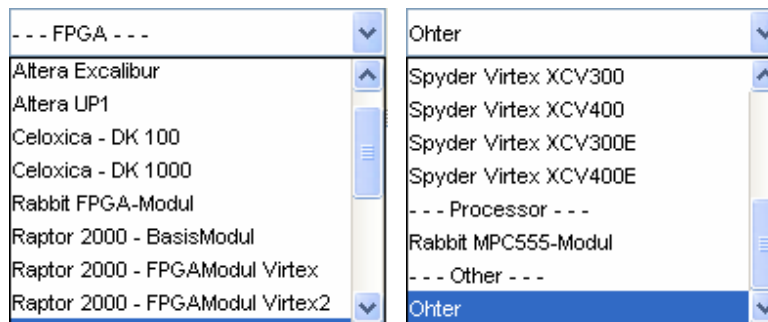


Abbildung A.13.: Board-Kategorie

Board-Kategorie

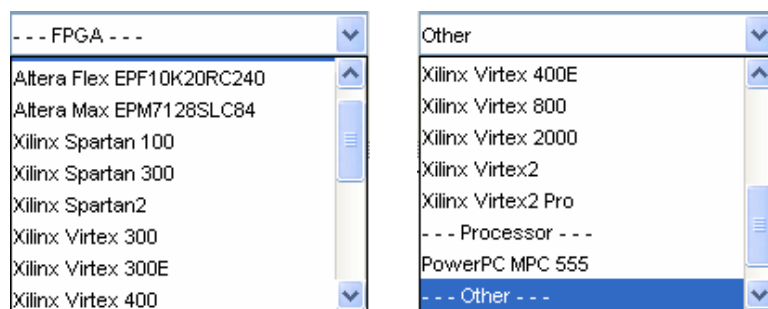


Abbildung A.14.: Chip-Kategorie

Task-Kategorie

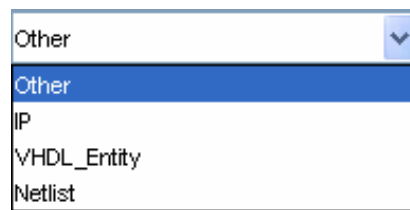


Abbildung A.15.: Task-Kategorie

Medium-Kategorie

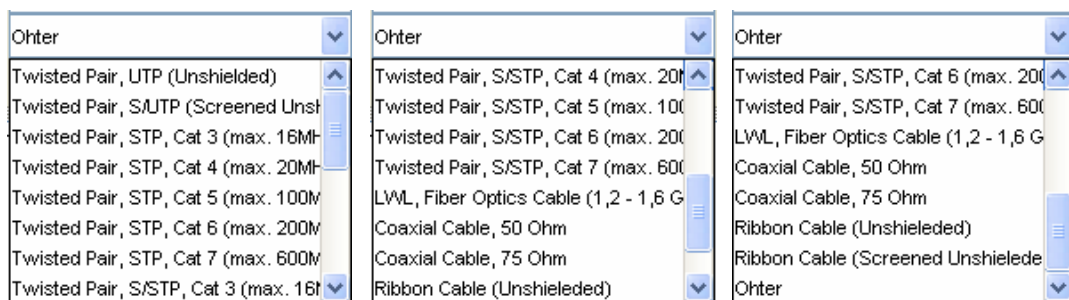


Abbildung A.16.: Medium-Kategorie

Danach wird die Instanz der Klasse “*Version*” für jede Komponente erzeugt. Die *Version* ist zur Angabe von Versionsinformationen vorgesehen. In dem Fall, dass keine Versionsinstanz für die Modellierten Systemkomponenten *Board*, *Chip*, *Task* und *Medium* vorhanden ist, werden diese aus der nächsten möglichen oberen Ebene der Systemhierarchie übernommen.

Interface-Klasse

Die Komponente “*sys1*” sowie ihre Subkomponenten wurden bereits modelliert. Für die Kommunikation zwischen diesen Komponenten sind Instanzen der Klasse Interface notwendig. Dazu werden alle, für die Kommunikation benötigten Interfaceinstanzen als Objekt in der Systemebene modelliert (vgl. Abbildung A.17). Die für die Kommunikation benötigten Signale werden als Attribute in der entsprechenden Interfaceinstanz deklariert. Eine Interface-Klasse unterscheidet sich von einem normalen Klassendiagramm in ihrem UML-Stereotyp. Die Auswahl “*interface*” steht für die Interface-Klasse als UML-Stereotyp zur Verfügung (siehe Abbildung A.17). Ebenso sind die “*package names*” zu beachten.

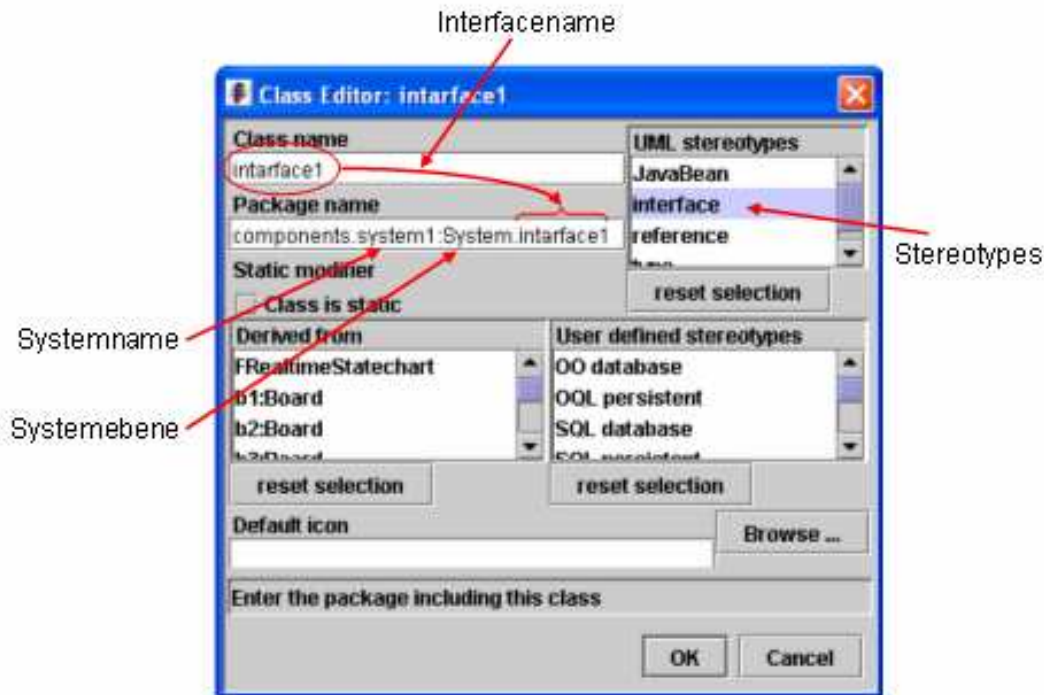


Abbildung A.17.: Interface Klassendeklaration

Die *Identification* und die *Properties* werden als Attribute der Interface-Klasse eingegeben. Alle benötigten Signale für die Kommunikation werden als Attribute mit einem initialen Wert deklariert (siehe Abbildung A.18). Dazu muss zuerst eine Klasse "Signal" modelliert werden, um die Signaltypen zu spezifizieren.

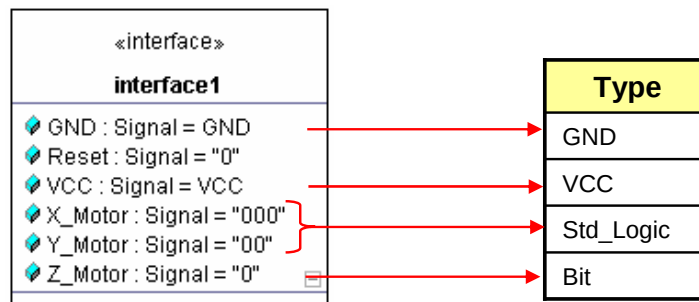


Abbildung A.18.: Signaldeklaration in der Interface-Klasse

Zusätzlich zu der Signaldeklaration gibt es eine Interface-Klasse, wie in Abbildung ?? dargestellt, mit den möglichen Eingabewerten für die Attribute. Die möglichen Werte und Beispieleingaben für die Attributinstanz stehen in den Tabellen der Abbildung A.19.

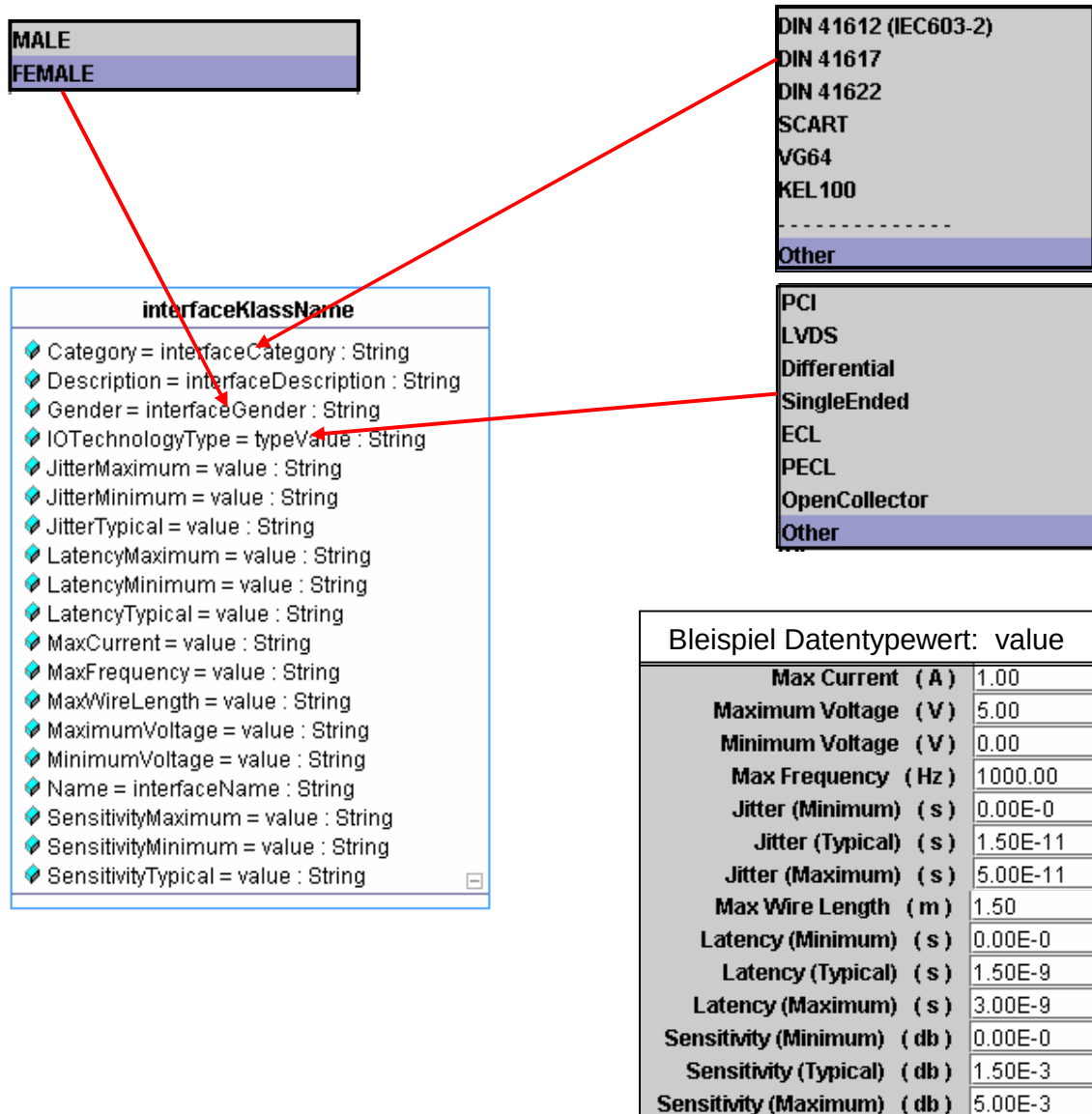


Abbildung A.19.: Mögliche Attributwerte der Interface-Klasse

Die meisten dieser Datentypen wurden aus den bereits für das IPQ Transfer Format entworfenen Datentypen übernommen [Vis02a, Vis02b].

Schnittstelle

Im Folgenden wird erläutert, wie die Kommunikation zwischen den Systemkomponenten (siehe Abbildung A.11) modelliert werden kann. Zwei Kommunikationsverbindungen sollen zwischen den Komponenten erstellt werden. Die erste Kommunikationsverbindung ist die Kommunikation zwischen dem Medium “*m1*” und “*m2*”. Die zweite Kommunikationsverbindung ist die Kommunikation zwischen den beiden Tasks “*task1*” und “*task2*”. Zuerst wird eine Schnittstelle (Port) für die Kommunikation an jeder Kommunikationskomponente erstellt. Dies geschieht

mit einem Linksklick auf die entsprechende Komponente (in diesem Fall: task1). Ein kleines Auswahlfenster (siehe Abbildung A.20) mit zwei Auswahlmöglichkeiten wird angezeigt. Die erste Auswahl dient zu der Erstellung von Subkomponenten und ist bereits erwähnt worden.

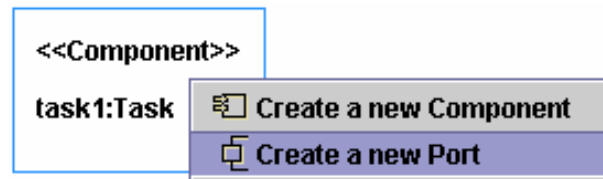


Abbildung A.20.: Erzeugung eines Ports

Bei Aktivierung von “Create a new Port” wird ein neues Eingabefenster angezeigt, in dem der Name des Ports eingegeben werden kann (siehe Abbildung A.21).

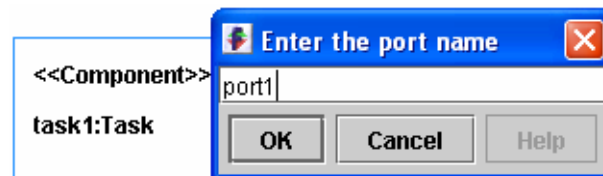


Abbildung A.21.: Auswahl des Portnamens

Mit der Bestätigung der Eingabe wird ein neuer Port mit dem Namen “port1” generiert (siehe Abbildung A.22).



Abbildung A.22.: Beispiel-Port

Nachdem *Port1* generiert wurde, werden automatisch die dazu gehörenden Objektdiagramme “*Port1:Port*” und Zustandsautomaten (“*Realtime Statecharts*”) als Referenz erzeugt (siehe Abbildung A.23, A.24).



Abbildung A.23.: Portinstanz

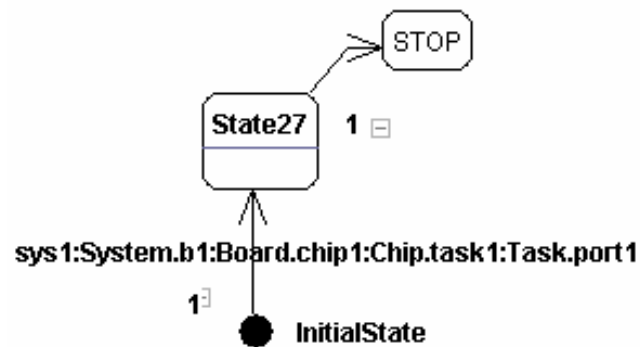


Abbildung A.24.: Zustandsautomaten für den Port “port1”

Die Attribute des IFSPorts (siehe Abbildung 4.13) werden in der entsprechenden Port-Instanz in diesem Fall “Port1:Port” instanziiert. Die Abbildung A.25 stellt die möglichen Werte, die bei der Attributdeklaration verwendet werden können, dar. Der Zustandsautomat wird im nächsten Abschnitt behandelt.

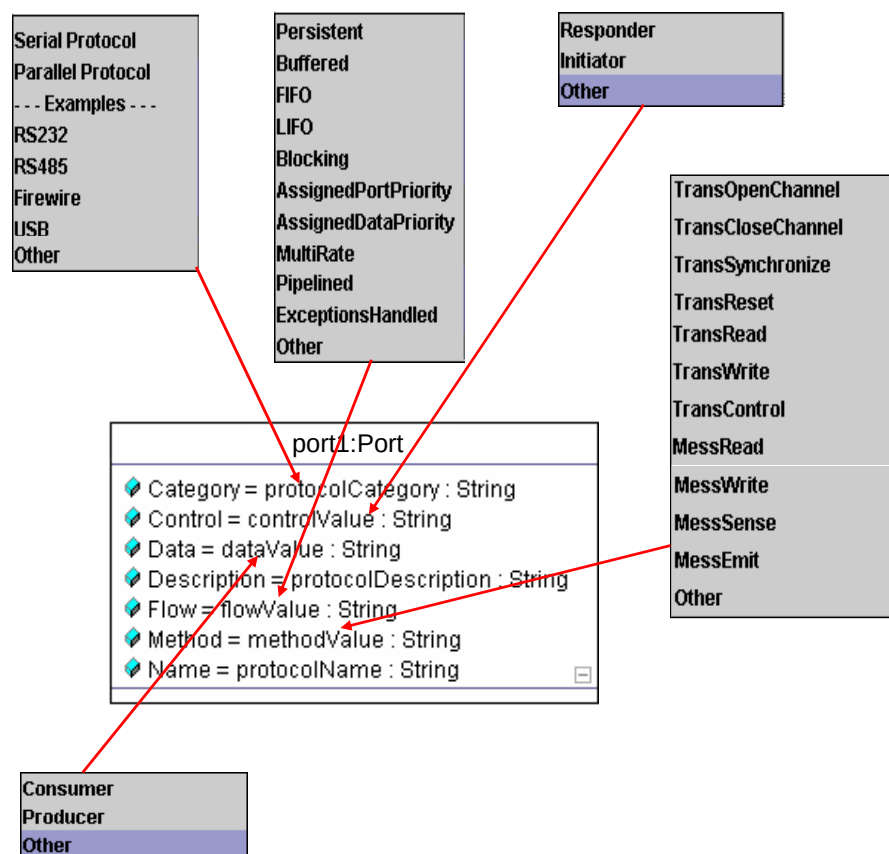


Abbildung A.25.: Attributeswerte einer Portinstanz

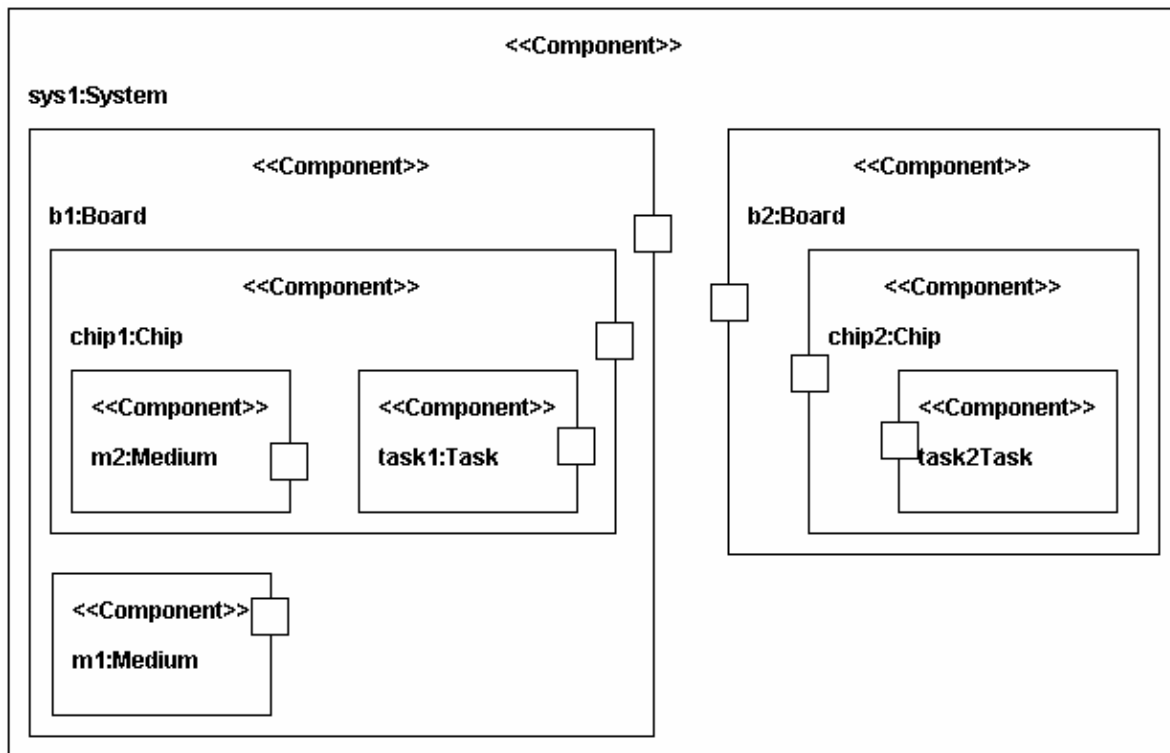


Abbildung A.26.: Komponenten und Ports der Systemarchitektur

Entsprechend werden alle Ports der Komponenten modelliert (siehe Abbildung A.26).

Bevor die Kommunikationskomponenten verbunden werden, müssen die Ports auf die entsprechende Interface-Instanz in der Systemebene referenzieren. Die Richtung für die Kommunikation wird nur auf der Modellierungsebene betrachtet. Für die Generierung des IFS-Formats kann die Richtung direkt aus dem Protokoll ermittelt. Mit einem Linksklick auf den entsprechenden Port wird ein Fenster mit zwei Auswahlmenüs angezeigt (siehe Abbildung A.27).

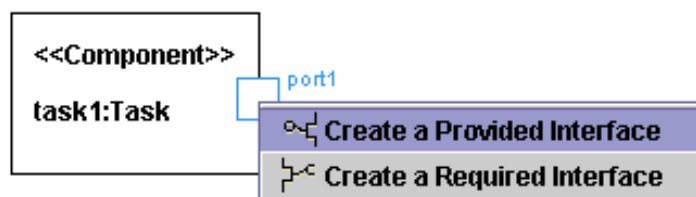


Abbildung A.27.: Erzeugen von Provided- oder Required Interfaces

Nach Auswahl von *Create a Provided* bzw. *Required Interface* wird ein weiteres Fenster mit allen vorhandenen modellierten Interface-Instanzen der Systemebene angezeigt (siehe Abbildung A.28).

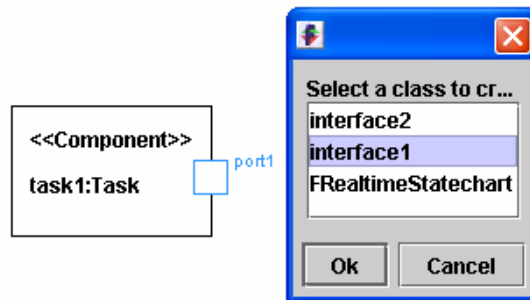


Abbildung A.28.: Auswahl des implementierten Interfaces

Mit der Bestätigung durch den OK-Knopf nach Auswahl der Interface-Instanz, wird der *Port* auf der entsprechenden Interface-Instanz referenziert, welche in diesem Fall *interface1* ist (siehe Abbildung A.29).

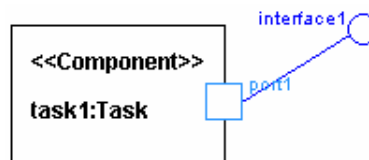


Abbildung A.29.: Referenzinterface

Ein *Provided Interface* wird bereits von *Port1* zur Verfügung gestellt. Für die Kommunikation zwischen Task “*task1*” mit dem Port “*port1*” und dem Chip “*chip1*” mit dem Port “*port2*” muss *port2* das gleiche Interface “*Interface1*” aber als *Required Interface* referenzieren (siehe Abbildung A.30).

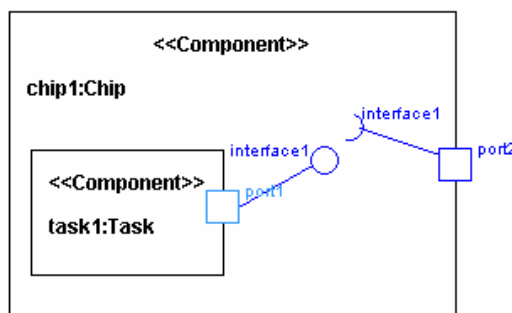


Abbildung A.30.: Provided und Required Interface

Als nächstes wird das *Provided* und *Required Interface* ausgewählt und durch einen *Connector* mit dem entsprechenden Icon verbunden (siehe Abbildung A.31).

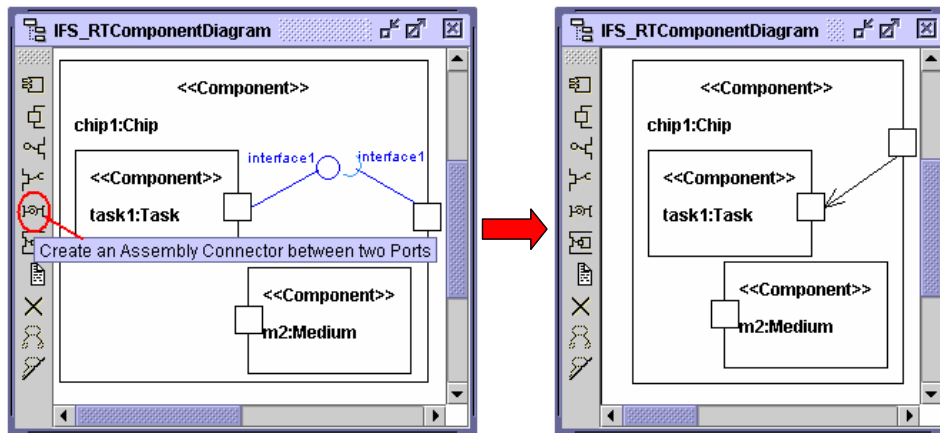


Abbildung A.31.: Connector

Entsprechend wird die gesamte Kommunikation erstellt. Die zu verbindenden *Ports* werden ausgewählt und durch *Connectoren* verbunden (siehe Abbildung A.32, A.33).

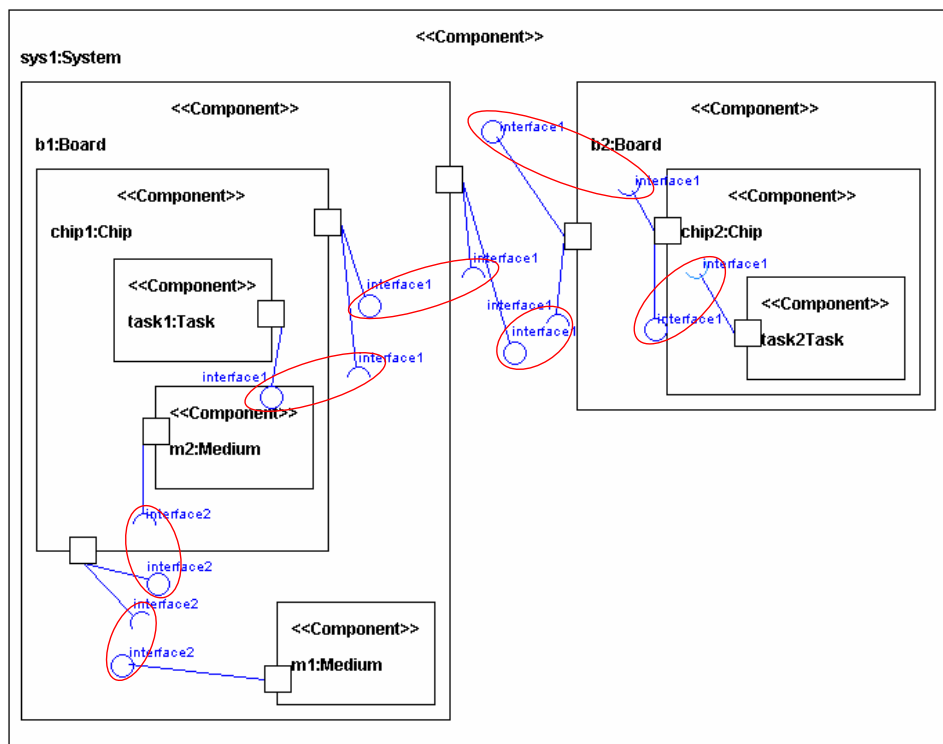


Abbildung A.32.: Erstellung der Connectoren

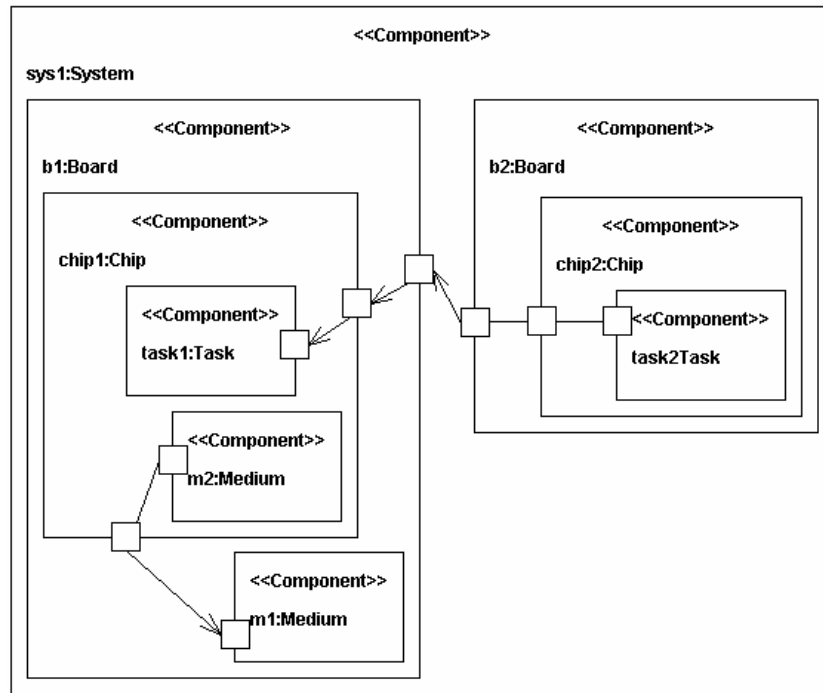


Abbildung A.33.: Gesamte Kommunikationsverbindungen

In diesem konkreten Beispiel sind die Interfaces *interface1* und *interface2* für beide Kommunikationen erzeugt worden. Die Interfaces *interface1* und *interface2* wurden vorher in der Systemebene als Objektdiagramme modelliert (siehe Abbildung A.34).

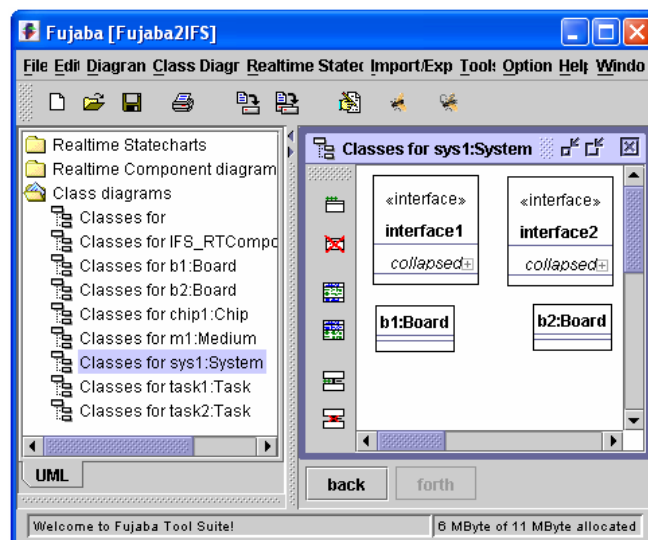


Abbildung A.34.: Systemebene-Klassendiagramme

A.1.3. Verhaltensbeschreibung

An dieser Stelle wird die Strukturbeschreibung des Systems „*sys1*“ durch Komponentendiagramme beschrieben. Das Verhalten für die Kommunikation zwischen den Komponenten wird als Zustandsautomat modelliert. Dazu werden die TPD und die Referenzsignale, die beide für die Protokollbeschreibung notwendig sind, als Instanz auf Chipebene bzw. Systemebene modelliert.

TPD

Die Abbildung A.35 stellt auf der linken Seite eine Tabelle dar. Die Tabelle beinhaltet die möglichen Werte für die Attribute Category und Unit des TPDs. Auf der rechten Seite ist die modellierte TPD-Instanz „*TPD_1*“ zu sehen. Die *TPD_1* wurde in der Chipebene „*Classes for chip1:Chip*“ (siehe Abbildung A.34) erzeugt.

TPD (Eingabeparameter)	
Category	Unit
FPGA	CLBs
PLA	Gates
ASIC	Memory
Processor	
Other	

TPD_1:TPD
- Category : String = FPGA - Quantity : Integer = 2 - Unit : String = Gates

Abbildung A.35.: TPD-Instanz

TPDClock

Ebenso wie die TPD-Instanz wird eine TPDClock-Instanz in der Chipebene modelliert (siehe Abbildung A.36). Die linke Tabelle stellt ein Beispiel für die Attributwerte des TPDClocks dar.

Frequency (Hz)	1.0E4
Skew Maximum (s)	3.00E-9
Skew Typical (s)	1.50E-9
Skew Minimum (s)	0.00E-0
Jitter Maximum (s)	5.00E-11
Jitter Typical (s)	1.50E-11
Jitter Minimum (s)	0.00E-0

TPDClock_1:TPDClock
- Description : String = Description to TPDClock_1 - Frequency : Integer = 1400 - JitterMax : String = 1.40E4 - JitterMin : String = 1.40E4 - JitterTyp : String = 1.40E4 - Name : String = TPDClock_1 - Network : String = Network - SkewMax : String = 1.40E4 - SkewMin : String = 1.40E4 - SkewTyp : String = 1.40E4

Abbildung A.36.: TPDClock-Instanz

ReferenceClock (Eingabeparameter)			
Phase	Type	StartValue	Frequency
0 Pi	Real Clock (TPD)	Logic One	ZB. 1.0E0
$\frac{1}{4}$ Pi	ReferenceClock	Logic Zero	
$\frac{1}{2}$ Pi			
$\frac{3}{4}$ Pi			

Tabelle A.1.: Eingabewert für die Attribute einer ReferenceClock

Referenzsignal

Die Elemente des Referenzsignals haben einen Attributname *TPDClockRefName*. Diese Attribute tragen den Namen der *TPDClock*-Instanz, die vorher in der Chipebene erzeugt wurden. In diesem Fall sind das *TPDClock_1* (siehe Abbildung A.36) und *TPDClock_2*.

- **ReferenceClock**

Die Tabelle A.1 stellt die möglichen Werte für die Attribute der *ReferenceClock* dar. In der Abbildung A.37 sind zwei *ReferenceClock*-Instanzen "*ReferenceClock_1*" und "*ReferenceClock_2*" zu sehen. Diese wurden in der Systemebene modelliert. Die *ReferenceClock_2* hat eine Phasenverschiebung von $\frac{1}{4}$ Pi zur *ReferenceClock_1*.

«type» ReferenceClock_2	«type» ReferenceClock_1
- Description : String = Description to ReferenceClock_2 - Frequency : String = 60.0E0 - Name : String = ReferenceClock_2 - Phase : String = 1/4 Pi - ReferenceClockName : String = ReferenceClock_1 - ReferenceClockType : String = Reference Clock - StartValue : Void = 0 - TPDClockRefName : String = TPDClock_2	- Description : String = Description to ReferenceClock_1 - Frequency : Integer = 1800 - Name : String = ReferenceClock_1 - StartValue : Integer = 1 - TPDClockRefName : String = TPDClock_1

Abbildung A.37.: ReferenceClock-Instanz mit und ohne Phasenverschiebung

- **TimeEvent**

Die Tabelle A.2 zeigt die möglichen Werte für das Attribut "*EventValue*" der *TimeEvent*-Klasse. Bei der Modellierung wird die Eingabe in der Kennung verwendet.

Zwei Instanzen der *TimeEvent*-Klasse werden ebenso auf der Systemebene erzeugt (siehe Abbildung A.38).

TimeEvent (Eingabe)	
EventValue	Kennung
Logic Zero	0
Logic One	1
High Peak	HP
Low Peak	LP
Signal Changed	SC

Tabelle A.2.: Mögliche Werte des Attributs “EventValue”

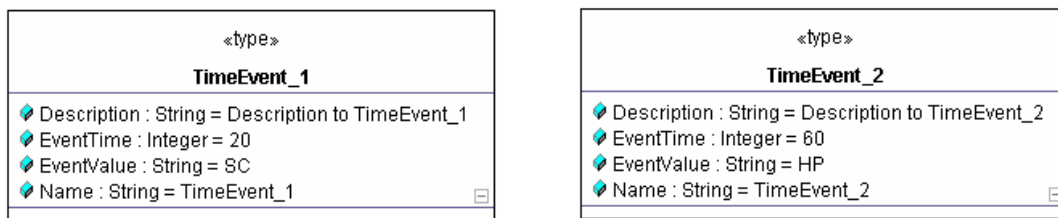


Abbildung A.38.: TimeEvent-Instanz

- **GlobalDate**

Eine Instanz der GlobalDate-Klasse “*GlobalDate_1*” wird auf der Systemebene modelliert (siehe Abbildung A.39). Das *GlobalDate_1* hat zwei Attribute vom Typ TimeEvent, das sind *TimeEvent1* und *TimeEvent2*. Die beiden Attribute haben als Instanz den Namen der vorher modellierten TimeEvent-Instanz (siehe Abbildung A.38). Die Namen werden für den Parseralgorithmus als Referenz verwendet. Mit diesen Namen werden die entsprechenden TimeEvents in der Systemebene gesucht.

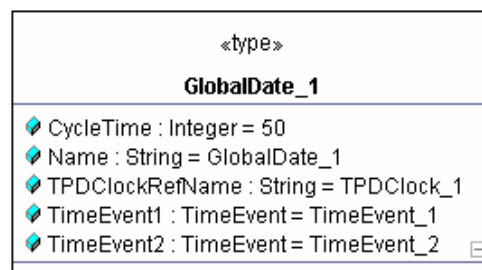


Abbildung A.39.: GlobalDate-Instanz

- **TimeOrDeadline**

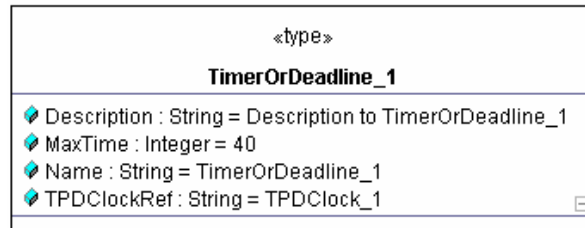


Abbildung A.40.: TimeOrDeadline-Instanz

A.1.4. Zustandsautomaten

Nachdem die *Ports* generiert wurden, werden automatisch die dazu gehörigen Zustandsautomaten (*“Realtime Statecharts”*) als Referenz erzeugt (siehe Abbildung A.24). Die Menüleiste *“Realtime Statechart”* (siehe Abbildung A.41) bietet mehrere Optionen für die Modellierung von Zustandsautomaten an. Die erste Option *“New/Edit Realtime State”* steht für die Erstellung bzw. Veränderung von Zuständen (State) und ihren Ausgaben (AutomataOutput) zur Verfügung. Die zweite Option *“New/Edit Realtime Transition”* ermöglicht die Darstellung sowie die Änderung von erstellten Transitionen. Die Option *Delete Realtime* bzw. *Delete RealtimeState* ist für die Löschung von bestehende Transitionen oder Zuständen da. Die weiteren Optionen wurden für diese Aufgabe nicht verwendet.

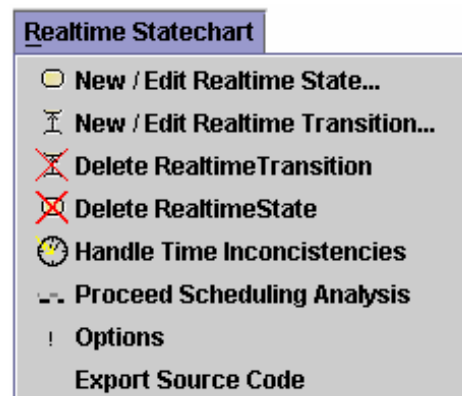


Abbildung A.41.: Auswahlmenü für das Editieren von Zustandsautomaten

State

Mit der Auswahl der Option *“New/Edit Realtime State”* wird ein neues Fenster geöffnet (siehe Abbildung A.42). Dort werden die Namen der Zustände *“State Name”* eingegeben. In diesem

TransitionType	Kennung
Asynchronous	A
Synchronous	S

ExpectedValue	Kennung
Low Value	0, LV
High Value	1, HV
Falling Edge	FE
Rising Edge	RE
Signal Changed	SC
Exceeded	E
Arrived	A

Tabelle A.3.: Mögliche Werte für die Attribute TransitionType und ExpectedValue

Fall handelt es sich um *state0*. Die *DoAction* ermöglicht die Eingaben des AutomataOutputs, dabei soll die in Kapitel 5.1 beschriebene Syntax eingehalten werden. Die Tabelle A.3 stellt die möglichen Werte für die Attribute TransitionType und ExpectedValue dar. Bei der Modellierung wird die dargestellte Abkürzung verwendet. In Abbildung A.42 befindet sich eine Tabelle. In dieser Tabelle werden die ExpectedValue-Werte unter dem Begriff *Value* dargestellt.

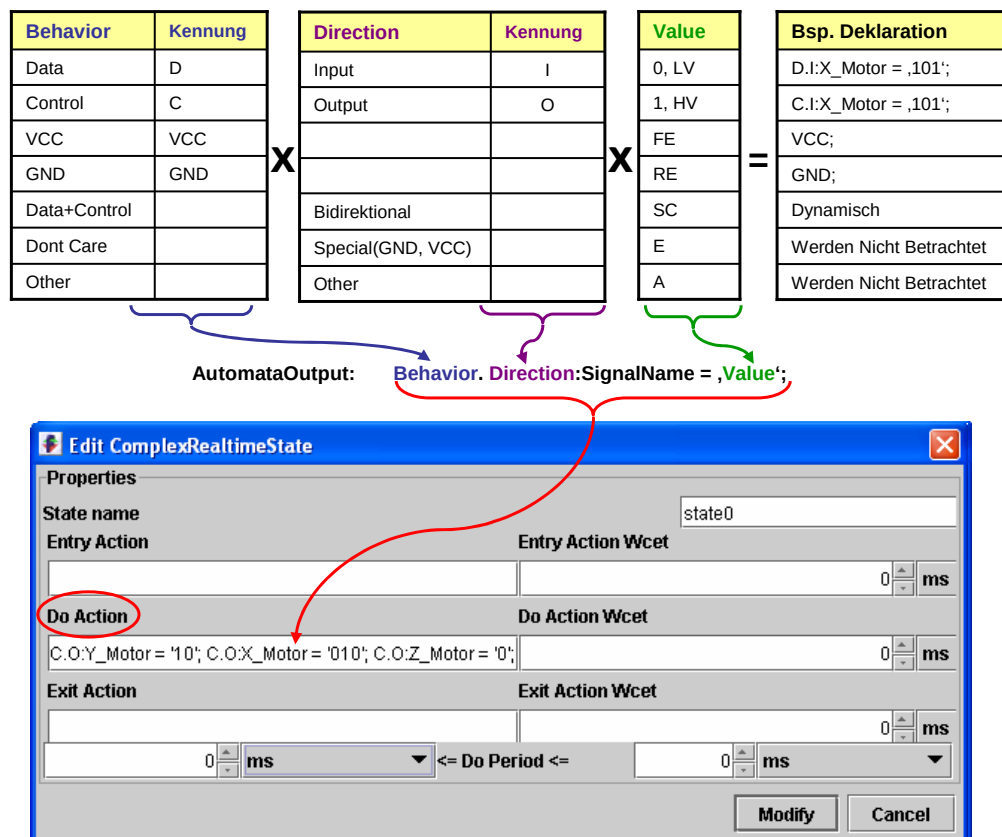


Abbildung A.42.: Eingabefenster für ein AutomataOutput: "Do Action"

Transition

Die Option “New / Edit Realtime Transition” bietet die Möglichkeit, mit Hilfe der in Abbildung A.43 dargestellten Fenster, die Transition sowie die Zustandsübergangsbedingungen zu modellieren. In diesem Fall handelt es sich um die Transition zwischen *state1* und *state0*. Bei dem Guard werden die Zustandsübergangsbedingungen für die Transition eingegeben. Ebenso wie bei dem AutomataOutput ist bei der Eingabe von Zustandsübergangsbedingungen, die in Kapitel 5.1 beschriebene Syntax zu beachten.

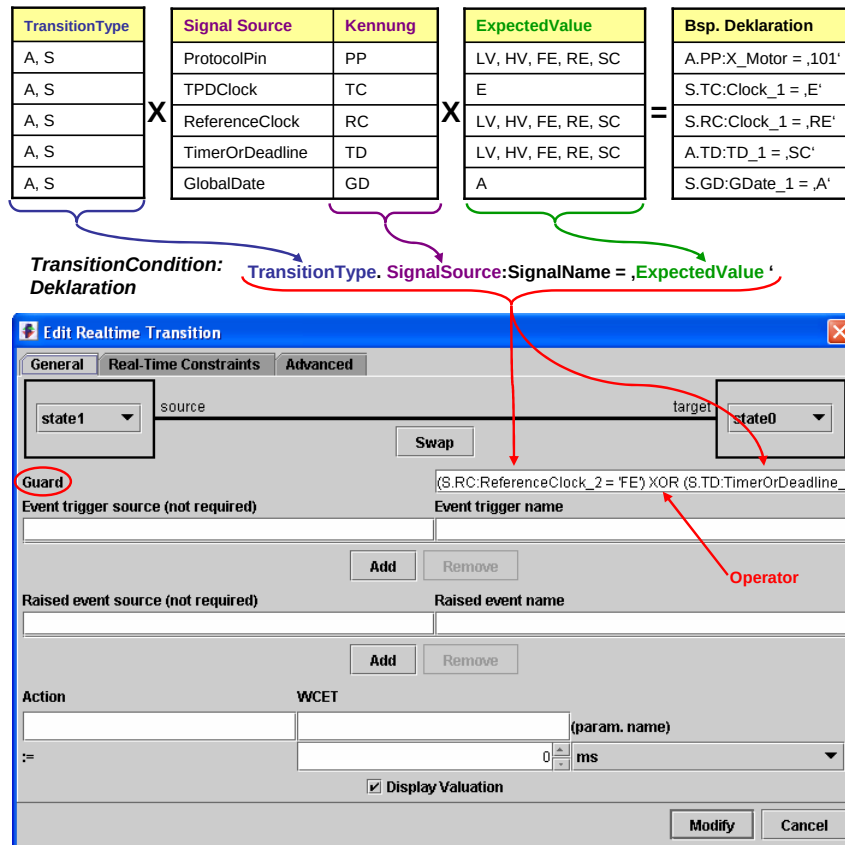


Abbildung A.43.: Eingabefenster für die Zustandsübergangsbedingungen “Guard”

Operator
AND
OR
XOR
NAND
NOR
XOR

Abbildung A.44.: Mögliche Werte für die Operatoren

Beispiel eines Zustandsautomaten

In Kapitel A.1.3 wurden die *ReferenceClock*-, die *TimerOrDeadline*-, sowie die *GlobalDate*- und die *TimeEvent*-Instanzen erzeugt. Als Beispiel für den Einsatz der in Kapitel 5.1 beschriebenen Syntax wurden all diese Instanzen bei der Modellierung des Zustandsautomaten des Ports “port1” in den Zustandsübergangsbedingungen verwendet (siehe Abbildung A.45).

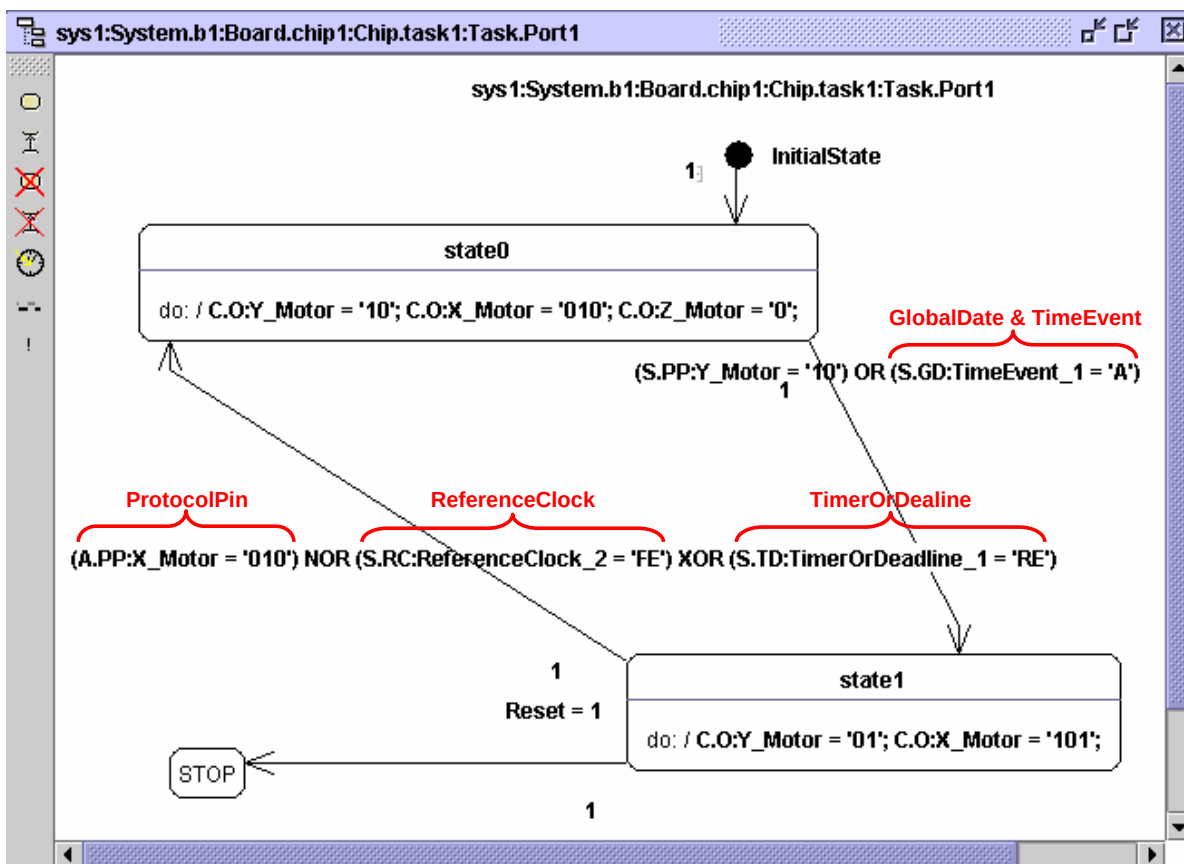


Abbildung A.45.: Modellierter Zustandsautomat für den Port “port1”

A.2. Beispiele zur XML Transformation

A.2.1. SAX-Parser-Beispiel

SaxPaserDemo [Bre01]

```
// Sax Interface
import org.xml.sax.XMLReader;

// Importiert die XMLReader-Implementation Ihres Herstellers
import org.apache.xerces.parsers.SAXParser; import
java.io.IOException; // Ausnahme bei dem ladung der Datei import
org.xml.sax.SAXException; // Problemen beim Parsen des Dokumentents
/**
 * SaxParserDemo erwartet eine XML-Datei und parst diese mittels SAX
 */
public class SaxPaserDemo {
    // Die Datei wird mittels URI(uri) übergeben

    public void performDemo(String uri){
        System.out.println("XML-Datei_wird_geparst:_ " + uri + "\n");
        try{
            // SAX Parser instanziiieren
            XMLReader parser = new SAXParser();
            // Das Dokument parsen
            MyContentHandler myContentHandler = new myContentHandler();
            // Setzen des implementierten Interfaces(ContentHandler)
            parser.setContentHandler(myContentHandler);
            parser.parse(uri);
        }catch (IOException e){
            System.out.println("Fehler_beim_Lesen_des_URI:_ "
                               + e.getMessage());
        }catch (SAXException e){
            System.out.println("Fehler_beim_Parsen:_ " + e.getMessage());
        }
    }
}

// Demoprogramm
public static void main(String[] args){
    SAXParserDemo parserDemo = new SAXParserDemo();
    // Das URI Oder URL soll mit dem entspreschende
    //Application Aufgelöst werden
    parserDemo.performDemo(uri);
}
}
```

A. Anhang

In der Klasse `MyContentHandler` werden die Methoden der Interfaceklasse `ContentHandler` implementiert, und ein entsprechender Konstruktor erzeugt. Die Methode ermöglicht die Bearbeitung des XML Dokuments mit der URI (`uri`). Die Methode definiert, was mit bestimmten vorgefundenen Elementen passieren soll und wie bestimmte Attribute behandelt werden.

```
import org.xml.sax.ContentHandler; import org.xml.sax.Attributes;
import org.xml.sax Locator;

/**
 * MyContentHandler implementiert das SAX-Interface
 */
class MyContentHandler implements ContentHandler{

    public void setDocumentLocator(org.xml.sax.Locator locator)
    {
    }

    public void startDocument() throws org.xml.sax.SAXException
    {
        // do something on start
    }

    public void endDocument() throws org.xml.sax.SAXException
    {
        // do something on end
    }
}
```

A.2.2. DOM-Parser-Beispiel

DOMPaserDemo [Bre01]

Diese Beispiel soll zeigen, wie ein DOM-Parser-Programm geschrieben werden kann, um XML Dokumente einzulesen. Dazu muss die DOM-Parser Klasse explizit importiert und instanziiert werden.

```
//DOM-Parser importieren
import org.apache.xerces.parsers.DOMParser; import
org.w3c.dom.Document; import org.w3c.dom.Node;
/**
 * Liest eine XML-Datei und bearbeiten diese und zeigt sie
 * mittels DOM an
 */
public class DOMPaserDemo {
    // URI der zu passenden Datei
```

```

public void performDemo(String uri){
    System.out.println("XML-Datei_wird_geparst:_ " + uri + "\n\n");
    // DOM-Parser instantieren
    DOMParser parser = new DOMParser();
    try{
        /**
         * Das Dokument parsen. Hier gibt es unterschiedlichen
         * DOM-Implementierungen der Methode parse() auf org.w3c.dom.
         * Die Methode liefert entweder ein Document-Objekt zurück
         * oder stellt eine Methode getDocument() zur Verfügung.
         */
        parser.parse(uri);
        Document docu = parser.getDocument();
        // Den eingelesenen DOM-Baum ausgeben
        printNode(docu);
    }catch (Exception e){
        System.out.println("Fehler_beim_Lesen_des_URI:_ " + e.getMessage());
    }
}

// Die Interface Methode können überschrieben werden um
// die DOM-Struktur zu verändern oder das Dokument
// als DOM-Node-Objekt verwenden.
public void printNode(Node node){
    // Den Typ des Knotens bestimmen
    // den Knoten bearbeiten und ausgeben
    // rekursiv auf den Kindknoten arbeiten
}

// Demoprogramm
public static void main(String[] args){
    DOMParserDemo parserDemo = new DOMParserDemo();
    // Das URI Oder URL soll mit dem entspreschende
    // Application aufgelöst werden
    parserDemo.performDemo(uri);
}
}

```

A.3. Java Quellcode

A.3.1. Synch2System()

```

public SystemArchitecture synch2System(Object obj)
{
    Component componentSystem = (Component) obj;
    SystemArchitecture system = new SystemArchitecture();
    // Key: Fujaba connector.getID(), Contend: Connector
    Hashtable sysConnectorHash = new Hashtable();
    // Key: Fujaba port.getID(), Contend: FujabaPort
    Hashtable portOfSubCompHash = new Hashtable();

    UMLClass umlClass = componentSystem.getComponentClass();
    // ----- Iterator of all attributes to System -----
    Iterator iterator = umlClass.iteratorOfAttrs();
    while (iterator.hasNext())
    {
        UMLAttr umlAttr = (UMLAttr) iterator.next();
        String umlAttrName = (umlAttr.getName()).replaceAll("_", "");
        String initialValue = umlAttr.getInitialValue();
        // Identification
        if (umlAttrName.equalsIgnoreCase("Name"))
        {
            system.setName(initialValue);
        }
        else if (umlAttrName.equalsIgnoreCase("Description"))
        {
            system.setDescription( initialValue );
        }
    }
    // ----- Iterator of all Subcomponents to System -----
    Iterator iter = componentSystem.iteratorOfSubcomponents();
    while (iter.hasNext())
    {
        Component subComponent = (Component) iter.next();

        String componentClassName =
            FujabaTools.extractClassName(subComponent.getFullName());
        String componentInstanceName =
            FujabaTools.extractInstanceName(subComponent.getFullName());
        // Version
        if (componentClassName.equalsIgnoreCase(
            StandardValues.version.getTag()))
        {
            system.setVersion(synch2Version(subComponent));
        }
    }
}

```

```

}
// BoardList
else if (componentClassName.equalsIgnoreCase(
    StandardValues.board.getTag()))
{
    system.addBoard(synch2Board(subComponent));
}
// MediumList
else if (componentClassName.equalsIgnoreCase(
    StandardValues.medium.getTag()))
{
    Object[] arrayObjt = new Object[3];
    arrayObjt[0] = subComponent;
    arrayObjt[1] = tpdClockHash;
    arrayObjt[2] = "-1";

    system.addMedium(synch2Medium(arrayObjt));
}
// Iterator of all Connections of one sub-component
Iterator iteratorOfConnectors =
    subComponent.getRealtimeComponentDiagram().iteratorOfConnector();
while (iteratorOfConnectors.hasNext())
{
    Connector connector = (Connector) iteratorOfConnectors.next();
    // Eintragen alles Connector in Hashtable
    if (!sysConnectorHash.containsKey(connector.getID()))
    {
        sysConnectorHash.put(connector.getID(), connector);
    }
    else
    {
        System.out.println(FujabaTools.getTab(3) + "Skip:_Connector_" +
            + connector.getName() + "\"_mit_ID_" + connector.getID()
            + "\";_already_in_Hashtable.");
    }
} // while
Iterator iterOfPort = subComponent.iteratorOfPorts();
while (iterOfPort.hasNext())
{
    Port port = (Port) iterOfPort.next();
    if (!portOfSubCompHash.containsKey(port.getID()))
        portOfSubCompHash.put(port.getID(), port);
}
}

Enumeration enum = sysConnectorHash.keys();

```

```

while (enum.hasMoreElements())
{
    String connectorID = (String) enum.nextElement();
    Connector connector = (Connector) sysConnectorHash.get(connectorID);

    Port portFrom = connector.getPortFrom();
    Port portTo = connector.getPortTo();
    if (portOfSubCompHash.containsKey(portFrom.getID())
        && portOfSubCompHash.containsKey(portTo.getID()))
    {
        de.upb.ipl.ifds.components.Interface interA = (de.upb.ipl.ifds .
            components.Interface) ifRefHashtable.get(portFrom.getID());
        de.upb.ipl.ifds.components.Interface interB = (de.upb.ipl.ifds .
            components.Interface) ifRefHashtable.get(portTo.getID());
        // InterfaceMapList
        try
        {
            if (interA.getParent() != null && interB.getParent() != null)
            {
                InterfaceMap interMap = new InterfaceMap(interA, interB);
                system.addInterfaceMap(interMap);
            }
        }
        catch (Exception e)
        {
            if (DEBUG > 0)
                System.out.println(FujabaTools.getTab(3) + "FATAL_ERROR:_ "
                    + "Program_code_wrong_in_InterfaceMap.java\n(method:_ "
                    + "InterfaceMap(Interface_interA,_Interface_interB))._"
                    + "At_least_one_parent_was_null.");
        }
    }
} // while
return system;
} // synch2System

```

A.3.2. synch2Protocol()

```

public Protocol synch2Protocol(Object obj)
{
    Object[] arrayObjVonInterAndPort = (Object[]) obj;
    de.upb.ipl.ifds.components.Interface x4jInterface =
        (de.upb.ipl.ifds.components.Interface) arrayObjVonInterAndPort[0];
    Port port = (Port) arrayObjVonInterAndPort[1];
    // Für ein Protocol key = UMLType(Name), value = tpdClock
    Hashtable tpdClockHashToPro = (Hashtable) arrayObjVonInterAndPort[2];

```

```

String isTpdClock = (String) arrayObjVonInterAndPort[3];
InterfaceProtocolFactory interProtocolFactory =
    new InterfaceProtocolFactory();
interProtocolFactory.generateProtocolFromInterface(x4jInterface);
Protocol generatedProtocol = interProtocolFactory.getGeneratedProtocol();
ProtocolMap generatedProtocolMap =
    interProtocolFactory.getGeneratedProtocolMap();
Hashtable portNameToProtPinListHashtable =
    interProtocolFactory.getPortRefIdToProHashtable();
//      ReferenceSignals
ReferenceSignals referenceSignals = new ReferenceSignals();
//      Für ein Protocol key = UMLType(Name), value = referenceClock
Hashtable referenceClockHashToPro = new Hashtable();
//      Für ein Protocol key = UMLType(Name), value = timerOrDeadline
Hashtable timerOrDeadlineHashToPro = new Hashtable();
//      Für ein Protocol key = UMLType(Name), value = timeEvent
//      Hashtable timeEventHashToPro = new Hashtable();
//      Key: globalDate_Name,
//      Contend: Object[3]/(CycleTime, TimeEvent, TPDClkref)
Hashtable globalDateHashToPro = new Hashtable();
//      Key: UMLType(Name), Contend: referenceClock oder timerOrDeadline
Hashtable rcToTeHashToPro = new Hashtable();
boolean refSignals = false;
//      Eintragen in protocolMapRefHashtable
if (!protocolMapRefHashtable.containsKey(port.getName()))
{
    protocolMapRefHashtable.put(port.getName(), generatedProtocolMap);
}

UMLClass portClass = port.getPortClass();
// ----- Iterator of all attributes to Protocol -----
Iterator iterator = portClass.iteratorOfAttrs();
while (iterator.hasNext())
{
    UMLAttr umlAttr = (UMLAttr) iterator.next();
    String umlAttrName = (umlAttr.getName()).replaceAll("_", "");
    String initialValue = umlAttr.getInitialValue();
    UMLType attrType = umlAttr.getUMLType();
    String typeName = attrType.getName();
    Object[] arrayObj = new Object[4];
    //      Identification
    if (umlAttrName.equalsIgnoreCase("Name"))
    {
        //      aus generateProtocolFromInterface()
    }
    if (umlAttrName.equalsIgnoreCase("Category"))

```

```

{
    generatedProtocol.setCategory(initialValue);
}
// Control
if (umlAttrName.equalsIgnoreCase("Control"))
{
    generatedProtocol.setControl(initialValue);
}
// Data
else if (umlAttrName.equalsIgnoreCase("Data"))
{
    generatedProtocol.setData(initialValue);
}
// Flow
else if (umlAttrName.equalsIgnoreCase("Flow"))
{
    generatedProtocol.setFlow(initialValue);
}
// TransactionType
else if (umlAttrName.equalsIgnoreCase("Method"))
{
    TransactionType transactionType = new TransactionType();
    transactionType.setTransactionMethod(initialValue);
    generatedProtocol.addTransactionType(transactionType);
}
// ReferenceClock
else if (typeName.startsWith("Ref") || typeName.startsWith("ref"))
{
    UMLClass umlClassRefSgl = (UMLClass) attrType;
    ReferenceClock referenceClock;
    // ReferenceClock ist schon in der gesamte Hashtable verfügbar
    if (referenceClockHash.containsKey(typeName))
    {
        // ReferenceCloc ist noch nicht in ReferenceSignals eingefügt
        if (!referenceClockHashToPro.containsKey(typeName))
        {
            referenceClock = (ReferenceClock) ((ReferenceClock)
                eferenceClockHash.get(typeName)).clone();
            if (isTpdClock.equalsIgnoreCase("-1"))
            {
                referenceClock.setTPDClockIDReference("0");
            }
            referenceClock.setID((
                referenceSignals.getReferenceClockList()).getFreeID());
            referenceSignals.addReferenceClock(referenceClock);
            referenceClockHashToPro.put(typeName, referenceClock);
        }
    }
}

```



```

        rcToTeHashToPro.put(typeName, referenceClock);
    }
}
else
{
    arrayObj[0] = tpdClockHashToPro;
    arrayObj[1] = umlClassRefSgl;
    arrayObj[2] = referenceClockHashToPro;
    arrayObj[3] = isTpdClock;
    referenceClock = synch2ReferenceClock(arrayObj);
    referenceClock.setID((
        referenceSignals.getReferenceClockList()).getFreeID());
    referenceSignals.addReferenceClock(referenceClock);
    // Einfügen der referenceClock in die
    // referenceClockkHashToPro (To ein Protocol)
    referenceClockHashToPro.put(typeName, referenceClock);
    rcToTeHashToPro.put(typeName, referenceClock);
    // Einfügen der referenceClock in die referenceClockkHash
    // (Für alles Protocol für die wieder verwendung)
    referenceClockHash.put(typeName, referenceClock);
}
refSignals = true;
}
// TimerOrDeadline
else if (typeName.startsWith("Ti") || typeName.startsWith("ti"))
{
    UMLClass umlClassRefSgl = (UMLClass) attrType;
    TimerOrDeadline timerOrDeadline;
    if (timerOrDeadlineHash.containsKey(typeName))
    {
        if (!timerOrDeadlineHashToPro.containsKey(typeName))
        {
            timerOrDeadline = (TimerOrDeadline) ((TimerOrDeadline)
                timerOrDeadlineHash.get(typeName)).clone();
            timerOrDeadline.setID(
                (referenceSignals.getTimerOrDeadlineList()).getFreeID());
            referenceSignals.addTimerOrDeadline(timerOrDeadline);
            timerOrDeadlineHashToPro.put(typeName, timerOrDeadline);
            rcToTeHashToPro.put(typeName, timerOrDeadline);
        }
    }
}
else
{
    arrayObj[0] = tpdClockHashToPro;
    arrayObj[1] = umlClassRefSgl;
    arrayObj[3] = isTpdClock;

```

```

    timerOrDeadline = synch2TimerOrDeadline(arrayObj);
    timerOrDeadline.setID(
        (referenceSignals.getTimerOrDeadlineList()).getFreeID());
    referenceSignals.addTimerOrDeadline(timerOrDeadline);
    timerOrDeadlineHashToPro.put(typeName, timerOrDeadline);
    rcToTeHashToPro.put(typeName, timerOrDeadline);
    timerOrDeadlineHash.put(typeName, timerOrDeadline);
}
refSignals = true;
}
// GlobalDate
else if (typeName.startsWith("Gl") || typeName.startsWith("gl"))
{
    UMLClass umlClassRefSgl = (UMLClass) attrType;
    Object[] arrayCtTpdclkTeventRef;
    X4JVector timeEventList = new X4JVector(
        StandardValues.referenceSignalsTimeEventList.getTag());
    // Schon in andere Protocol vorhanden
    if (globalDateClassHash.containsKey(typeName))
    {
        // Noch nicht in diese Protocol Vorhanden
        if (!globalDateHashToPro.containsKey(typeName))
        {
            arrayCtTpdclkTeventRef =
                (Object[]) globalDateClassHash.get(typeName);
            referenceSignals.setCycleTime(
                (String) arrayCtTpdclkTeventRef[0]);
            referenceSignals.setTPDClockIDReference(
                (String) arrayCtTpdclkTeventRef[1]);
            timeEventList = (X4JVector) arrayCtTpdclkTeventRef[2];
            referenceSignals.setTimeEventList(timeEventList);
            globalDateHashToPro.put(typeName, arrayCtTpdclkTeventRef);
        }
    }
}
else
{
    arrayObj[0] = tpdClockHashToPro;
    arrayObj[1] = umlClassRefSgl;
    arrayObj[2] = rcToTeHashToPro;
    arrayObj[3] = isTpdClock;
    arrayCtTpdclkTeventRef = synch2GlobalDate(arrayObj);
    referenceSignals.setCycleTime(
        (String) arrayCtTpdclkTeventRef[0]);
    referenceSignals.setTPDClockIDReference(
        (String) arrayCtTpdclkTeventRef[1]);
    timeEventList = (X4JVector) arrayCtTpdclkTeventRef[2];
}

```

```

        referenceSignals.setTimeEventList(timeEventList);
        globalDateHashToPro.put(typeName, arrayCtTpdclkTeventRef);
        globalDateClassHash.put(typeName, arrayCtTpdclkTeventRef);
    }
}
refSignals = true;
}
if (refSignals)
    generatedProtocol.setReferenceSignals(referenceSignals);
// Version (Version from Task)
// ----- Iterator of all Subcomponent from Task -----
Iterator iter = (port.getComponent()).iteratorOfSubcomponents();
while (iter.hasNext())
{
    Component subComponent = (Component) iter.next();
    String componentClassName =
        FujabaTools.extractClassName(subComponent.getFullName());
    String componentInstanceName =
        FujabaTools.extractInstanceName(subComponent.getFullName());
    // Version
    if (componentClassName.equalsIgnoreCase(
        StandardValues.version.getTag()))
    {
        generatedProtocol.setVersion(synch2Version(subComponent));
    }
}
}

ProtocolStateChart protocolStateChart = port.getProtocolStateChart();
// ----- Iterator of all State to Port -----
//      StateID, Statische Werte und MooreOutputs setzen
Iterator iterOfAllStates =
    protocolStateChart.iteratorOfAllUMLRealtimeStates();
int stateID = 1;
while (iterOfAllStates.hasNext())
{
    UMLRealtimeState umlState = (UMLRealtimeState) iterOfAllStates.next();
    try
    {
        UMLComplexRealtimeState complexUmlState =
            (UMLComplexRealtimeState) umlState;
        Object[] complexUmlStateUndX4jHash = new Object[2];
        complexUmlStateUndX4jHash[0] = complexUmlState;
        complexUmlStateUndX4jHash[1] = portNameToProtPinListHashtable;
        State x4jState = synch2State(complexUmlStateUndX4jHash);
        x4jState.setID((generatedProtocol.getStateList()).getFreeID());
        // Eintragen des State in die Hashtable "stateHashtable"
    }
}

```

```

        generatedProtocol.addState(x4jState);
        if (!stateHashtable.containsKey(complexUmlState.getID()))
        {
            stateHashtable.put(complexUmlState.getID(), x4jState);
        }
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}
//TransitionList auf State Setzen dafür nutzt die stateHashtable
Iterator iterOfAllStatesForTransition =
    protocolStateChart.iteratorOfAllUMLRealtimeStates();
while (iterOfAllStatesForTransition.hasNext())
{
    UMLRealtimeState umlState =
        (UMLRealtimeState) iterOfAllStatesForTransition.next();
    try
    {
        UMLComplexRealtimeState complexUmlState =
            (UMLComplexRealtimeState) umlState;
        // ----- Iterator of all Transition to State -----
        Iterator iterOfTransition =
            complexUmlState.iteratorOfUMLRealtimeTransition1();
        while (iterOfTransition.hasNext())
        {
            UMLRealtimeTransition transition =
                (UMLRealtimeTransition) iterOfTransition.next();
            Object[] transitionUndX4jHash = new Object[3];
            transitionUndX4jHash[0] = transition;
            transitionUndX4jHash[1] = portNameToProtPinListHashtable;
            transitionUndX4jHash[2] = rcToTeHashToPro;
            Transition x4jTransition = synch2Transition(transitionUndX4jHash);
            if (x4jTransition != null)
            {
                State x4jState =
                    (State) stateHashtable.get(complexUmlState.getID());
                x4jTransition.setID((x4jState.getTransitionList()).getFreeID());
                x4jState.addTransition(x4jTransition);
            }
        }
    }
}
catch (Exception e)
{
    System.out.println(e);
}

```

```

    }
  }
  return generatedProtocol;
} //synch2Protocol

```

A.3.3. extractType()

```

/*
 * Methode return String "Std_Logic" or "Bit" or "VCC" or "GND"
 */
public static String extractType(String initialValue)
{
    String pinValue = (getPinValue(initialValue, 1));
    if (pinValue.equalsIgnoreCase("0")
        || pinValue.equalsIgnoreCase("1"))
    {
        if (getAnzahlPin(initialValue) > 1)
            return "Std_Logic";
        else
            return "Bit";
    }
    else if (pinValue.equalsIgnoreCase("V"))
        return "VCC";
    else if (pinValue.equalsIgnoreCase("G"))
        return "GND";
    else
        return "Std_Logic";
} // extractType

```


Literaturverzeichnis

- [Alf99] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilerbau, 2. Teil*. R. Oldenbourg Verlag München Wien, Dezember 1999.
- [Alf02] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. 2002.
- [And04] Andreas Andresen. *Komponentenbasierte Softwareentwicklung mit MDA, UML 2 und XML*, volume 2. Auflage. Hanser fachbuchverlag edition, August 2004.
- [Bec03] Beckert, Bernhard. *Formale Spezifikation und Verifikation von Software*. Universität Koblenz-Landau, <http://www.uni-koblenz.de/~beckert/Lehre/Spezifikation-Verifikation/skriptum-schmitt.ps.gz>, Wintersemester 2002/2003.
- [Ber04] Bernd, Oestereich. *Objektorientierte Softwareentwicklung - Analyse und Design mit der UML 2*, volume 6. überarbeitete Auflage. R. Oldenbourg Verlag München Wien www.oose.de/uml, Februar 2004.
- [Boo] Booch, Grady and Rumbaugh, Jim and Jacobson, Ivar. *Das UML Benutzerhandbuch mit Bleispielen zunehmender Komplexität*.
- [Bor] Borland. *Object International. Together J, the Together J case tool*. Online at: <http://www.togethersoft.com>.
- [Bre01] Brett, McLaughlin. *Deutsche Übersetzung von Matthias Kalle Dalheimer, Java und Xml*, volume 1. Auflage. by o'reilly verlag edition, 2001.
- [Fic03] Fick, Oliver. Visualisierung von Parametern komplexer Schnittstellen für eingebettete Systeme auf Basis von XML. Master's thesis, Universität Paderborn, Warburger Str. 100, 33098 Paderborn, June 2003.
- [Hol03] Holger Giese and Matthias Tichy and Sven Burmester and Wilhelm Schäfer and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.
- [Ihm01] Ihmor, Stefan. Entwurf von Echtzeitschnittstellen am Beispiel interagierender Roboter. Master's thesis, Universität Paderborn, Warburger Str. 100, 33098 Paderborn, November 2001.

- [Ihm02] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. A Consistent Design Methodology for Configurable HW/SW-Interfaces in Embedded Systems. In *Proc. of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems*, Montreal, Canada, Aug. 2002.
- [Jec04a] Jeckle, Mario. *Unified Modeling Language (UML) Tools*. <http://www.jeckle.de/umltools.htm>, Juni, 2004.
- [Jec04b] Jeckle, Mario and Rupp, Chris and Hahn, Jürgen and Zengler, Barbara and Queins, Stefan. *UML2 glasklar*. www.uml-glasklar.de, carl hanser verlag münchen wien edition, 2004.
- [Pat03] Patrick Grässle, Henriette Baumann, Philippe Baumann. *UML projektorientiert - Ausblick auf den neuen Standard 2.0 - Galileo Computing*, volume 1. Auflage. Mai 2003.
- [Pat04] Patrick Grässle, Henriette Baumann, Philippe Baumann. *UML 2.0 projektorientiert - Geschäftsprozessmodellierung, IT-System-Spezifikation und Systemintegration mit der UML, mit Farbposter*, volume 3. Auflage. Juli 2004.
- [Rat] Rational. Rose. The Rational Rose case tool. <http://www.rational.com>.
- [Rat99] Rational Software Corporation. *UML documentation version 1.3*. Online at. <http://www.rational.com>, 1999.
- [Rei98] Reinhard Wilhelm, Dieter Maurer. *Übersetzerbau*, volume 2. überarbeitete und erweiterte Auflage. Springer, Berlin, Oktober 1998.
- [Sve04] Sven Burmester and Holger Giese and Jörg Niere and Matthias Tichy and Jörg P. Wadsack and Robert Wagner and Lothar Wendehals and Albert Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, August 2004.
- [Tho04] Thomas Erler. *Das Einsteigerseminar UML 2*. Vmi buch edition, November 2004.
- [Uni] Universität Paderborn. *Fujaba - From Uml to Java And Back Again*. <http://wwwcs.upb.de/cs/fujaba/>.
- [Uni04] Unified Modeling Language. *Superstructure version 2.0* Online at. <http://www.omg.org/docs/ptc/03-08-02.pdf>, Published on April 30, 2004.
- [Vir01] Virtual Component Transfer Development Working Group. *Virtual Component Transfer Specification Version 2.1*. VSI AllianceTM, vct 1 2.1 edition, January 2001.

- [Vis02a] Visarius, M. and Lessmann, J. and Kelso, F. and Hardt, W. and Amelunxen, C. and Scholand, A. and Ihmor, S. Definition of the IPQ Format with Respect to Specialized Description Features. Technical Report TR-IPL-2002-04, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, August 2002.
- [Vis02b] Visarius, Markus and Hardt, Wolfram and Ihmor, Stefan and Lessmann, Johannes. IPQ Format: Concepts and Implementation. In *Proc. of the Int. Workshop of the MEDEA+ Project A-511 ToolIP*, Madrid, Spain, January 2002.
- [W3C02] W3C. *Extensible Markup Language (XML) 1.0 - Deutsche Übersetzung*, volume 2. Auflage. <http://edition-w3c.de/TR/2000/REC-xml-20001006>, 20. Januar 2002.
- [Wik] Wikipedia. *Die freie Enzyklopädie*. Online at. <http://de.wikipedia.org/wiki/Hauptseite>.

Datenflussoptimierung in rekonfigurierbarer Hardware durch Pipelining

Christian Behler

Die Effizienz einer Anwendung sowie die Menge an benötigten Ressourcen sind speziell in eingebetteten Systemen von Bedeutung. Dies gilt selbstverständlich auch für die vorhandenen Schnittstellenadapter. Für transparente Kommunikationskomponenten sind insbesondere zwei Parameter von großen Interesse: die Latenz sowie die verbrauchte Chipfläche.

Der zweite Beitrag dieses Buches befasst sich daher mit der Optimierung dieser beiden Parameter für die Implementierung von Schnittstellenadaptern. Es werden zwei Optimierungsverfahren vorgestellt, die einerseits eine konsequente Nutzung von Pipelining und zum anderen die Rekonfigurierbarkeit der betrachteten Zielarchitektur gezielt auszunutzen.

Eine präzise Analyse des Datenflusses des betrachteten Protokolladapters (Interface Block) beschreibt das Potenzial, die Latenz der Protokollkonvertierung durch eine parallele Ausführung von Aktionen in Form einer Pipeline zu optimieren. Es werden zwei Lösungsverfahren für diese Probleme vorgestellt, das "Frame Based Schedule" und das "Subframe Based Schedule".

Orthogonal zu den beiden Verfahren zur Verringerung der Latenz wird ein Ansatz zur Optimierung des Flächenbedarfs präsentiert. Die betrachtete Zielplattform ist ein FPGA und kann also zur Laufzeit rekonfiguriert werden. Durch die Einführung von Rekonfigurationsschritten in den Ablauf unter Verwendung einer optimalen Caching-Strategie kann die benötigte FPGA-Fläche der Protokollkonvertierungen signifikant reduziert werden.

Inhaltsverzeichnis

1	Einführung	109
1.1	Motivation	109
1.2	Aufgabenstellung	111
2	Grundlagen	113
2.1	Das FPGA	113
2.2	Interface Block (IFB)	114
2.3	Datenfluss im IFB	116
2.4	IFD-Mapping	117
2.5	Ausführung als Pipeline	119
2.6	Detaillierte Darstellung des IFB Datenflusses	124
3	Datenflussoptimierung	127
3.1	Ausführung des IFB als Pipeline	127
3.2	Optimierungskonzept	129
3.3	Optimierungsverfahren	132
3.3.1	framebased schedule	133
3.3.2	subframebased schedule	135
3.4	Ergebnisse	137
4	Rekonfigurierte Ausführung	143
4.1	I-P-O Scheduling eines rekonfigurierbaren IFB	145
4.2	Ergebnisse	147
5	Zusammenfassung und Ausblick	153
5.1	Zusammenfassung	153
5.2	Ausblick	153
	Literaturverzeichnis	155

1 Einführung

1.1 Motivation

Kommunikation ist in vielen Bereichen ein wichtiger Faktor. Neu entwickelte und bereits bestehende Komponenten müssen in der Lage sein, Informationen untereinander auszutauschen. Dazu ist es wichtig, dass Komponenten, die miteinander kommunizieren, die gleiche Kommunikationsstruktur verwenden, damit die Informationen korrekt interpretiert und verarbeitet werden können.

Im Bereich der Informationstechnik wird häufig auf bereits bestehende Komponenten zurückgegriffen, um neue Funktionalitäten zu implementieren. Somit ist es oft schwierig, eine gemeinsame Kommunikationsstruktur zu finden, so dass die einzelnen Teile des Gesamtsystems miteinander kommunizieren können. Um dieses Problem zu umgehen, gibt es zwei mögliche Ansätze [Lok05]:

- Die verwendeten Komponenten benutzen die gleiche Kommunikationsstruktur und eine Anpassung ist nicht nötig. Dadurch wird das System unflexibel, da nicht beliebige Komponenten eingebunden werden können.
- Jede Komponente muss ein Interface bieten, dass mit einem standardisierten Bussystem kommunizieren kann. Diese Standardisierung kann aber dazu führen, dass Komponenten, die von der Funktionalität her recht klein sind, ein recht großes Interface implementieren müssen, um in das vorhandene System integriert werden zu können. Somit kann ein unverhältnismäßig hoher Aufwand entstehen.

Betrachtet man diese Problematik auf der Ebene der Eingebetteten Systeme, so existiert dort ein weiterer Lösungsansatz. Durch die sogenannte *Interface Synthesis* (IFS), einer Methodik für die Modellierung und automatisierte Synthese von Schnittstellen, kann ein Modul generiert werden, welches zwei Kommunikationspartner, zwischen denen eine direkte Kommunikation nicht möglich ist, verbindet. Dieses Modul wird *Interface Block* (IFB) genannt und fungiert als "Übersetzer" zwischen den inkompatiblen Komponenten. So wird der Austausch von Informationen ermöglicht, ohne dass die Kommunikationspartner modifiziert werden müssten.

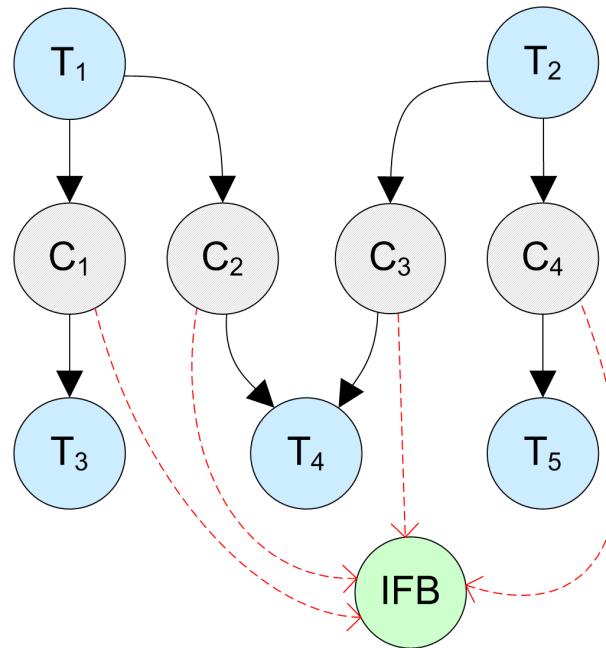


Abbildung 1.1: Kommunikationsgraph

Abbildung 1.1 zeigt einen möglichen Einsatz für den IFB. Es existieren fünf Kommunikationspartner (hier als Task bezeichnet), zwischen denen eine direkte Kommunikation nicht möglich ist. Task₁ will mit Task₃ und Task₄ kommunizieren, Task₂ will Daten an Task₄ und Task₅ senden. Zwischen jedes Kommunikationspaar beschreibt eine Abbildungsvorschrift (hier als Kommunikationsknoten C_{1..4} bezeichnet) wie die gelesenen Daten zu transformieren sind, so dass der Empfänger diese korrekt interpretieren kann. Eine Abbildungsvorschrift wird auf einen IFB abgebildet, d.h. sie wird dort implementiert. In Kapitel 2 wird der genaue Aufbau des IFBs und dessen Funktionalität erklärt. Es wird auch darauf eingegangen, welche Hardware verwendet werden kann, um die Funktionalität des IFBs zu realisieren.

Die Transformation der Daten im IFB verursacht eine *Latenz*, da das Senden und Empfangen der Daten sowie die Verarbeitung (also die "Übersetzung") eine gewisse Zeit benötigt. Latenz bezeichnet die Zeit, die vom Senden der Daten bis zum Empfangen der ersten Information vergeht. Daten, die zuerst von einem IFB verarbeitet werden, kommen also zu einem späteren Zeitpunkt bei den empfangenden Tasks, an als wenn die Daten direkt gesendet würden.

Ein weiterer Aspekt bei der Betrachtung des IFBs ist der physikalische Platz auf dem Chip, der benötigt wird, um die Komponenten des IFBs zu implementieren. Je größer ein IFB wird, desto mehr Platz benötigt dieser. Große Chips sind allerdings mit hohen Kosten verbunden. Es ist ein allgemeines Ziel, diese Kosten so gering wie möglich zu halten.

1.2 Aufgabenstellung

Ausgehend von der existierenden IFB-Technologie sollen im Rahmen dieser Studienarbeit zwei Verfahren zur Optimierung der Aspekte Latenz und Fläche untersucht werden.

Datenflussoptimierung In Kapitel 3 wird die Latenz durch Datenflussoptimierung reduziert. Dazu wird versucht, die maximal mögliche Anzahl an Eingabe-, Verarbeitungs- und Ausgabeprozessen parallel laufen zu lassen, damit die empfangenen Daten früher verarbeitet und zum Empfänger gesendet werden können. Um dies zu erreichen, werden die eingehenden Datenpakete betrachtet. Die tatsächliche Größe, die der IFB auf dem Chip, auf dem er implementiert ist, benötigt, wird nicht berücksichtigt.

Flächenoptimierung Der zweite Ansatz beschäftigt sich mit der optimalen Ausnutzung der Fläche auf dem Chip, auf dem der IFB implementiert ist. Die Grundlage ist ein Chip, der zu klein ist, um alle gewünschten Funktionalitäten gleichzeitig unterzubringen und es somit erforderlich macht, Funktionalitäten zur Laufzeit zu laden. Der Ansatz dazu wird in Kapitel 4 vorgestellt.

Diese beiden Ansätze sind *nicht* gleichzeitig realisierbar und zwar aus folgendem Grund: Der erste Ansatz berücksichtigt keine Restriktionen bezüglich der Größe des IFBs (und somit des Chips). Es wird davon ausgegangen, dass die vollständige IFB-Funktionalität zu jeder Zeit vorhanden ist. Der zweite Optimierungsansatz verletzt diese Restriktion: Die Flächenoptimierung setzt voraus, dass *nicht* die komplette Funktionalitäten zu jeder Zeit vorhanden sein muss. Das führt dazu, dass ungenutzte Funktionalitäten verdrängt und durch solche Funktionalitäten ersetzt werden müssen, die gerade benötigt werden. Eine Optimierung des Datenflusses, wie sie in Kapitel 3 beschrieben wird, ist dabei nicht möglich, da die Optimierung das Verdrängen und Laden von Funktionalitäten nicht berücksichtigt. Beide Verfahren werden anhand von ausgewählten Beispielen bewertet. Abschließend wird in Kapitel 5 eine kurze Zusammenfassung gegeben.

2 Grundlagen

In diesem Kapitel werden grundlegende Begriffe erklärt sowie eine Einführung in die verwendete Hardware und Software gegeben. Dabei wird insbesondere auf Field Programmable Gate Arrays (siehe Kapitel 2.1) als eine mögliche Zielplattform für IFBs eingegangen. Die partielle Rekonfigurierbarkeit dieser Hardwareplattform ist die Grundlage für die Flächenoptimierung, die in Kapitel 4 vorgestellt wird.

2.1 Das FPGA

Ein *Field Programmable Gate Array* (FPGA) ist ein programmierbarer Chip, der eine geeignete Ausführungsplattform für den IFB darstellt. Der Chip kann rekonfiguriert werden, d.h. Funktionalitäten können zu Laufzeit ausgetauscht werden. Somit kann das FPGA auch als Plattform für die rekonfigurierbare Version des IFBs [Ihm04] genutzt werden. Im Folgenden wird ein kurzer Einblick in die Funktionalität von FPGAs gegeben. Eine detailliertere Darstellung findet sich in Kapitel 4.

Ein FPGA besteht aus rekonfigurierbaren Einheiten, sogenannten *slices*. Mehrere *slices* ergeben einen Rekonfigurierungsblock (RB) oder *Slot*. In diese Slots können die gewünschten Funktionen (Programme) geladen werden. Allerdings benötigt jeder Slot sogenannte *Bus Macros*, die ihn mit den anderen Slots verbindet. Die Trennung durch Bus Macros ist für die Rekonfigurierung notwendig und technisch bedingt.

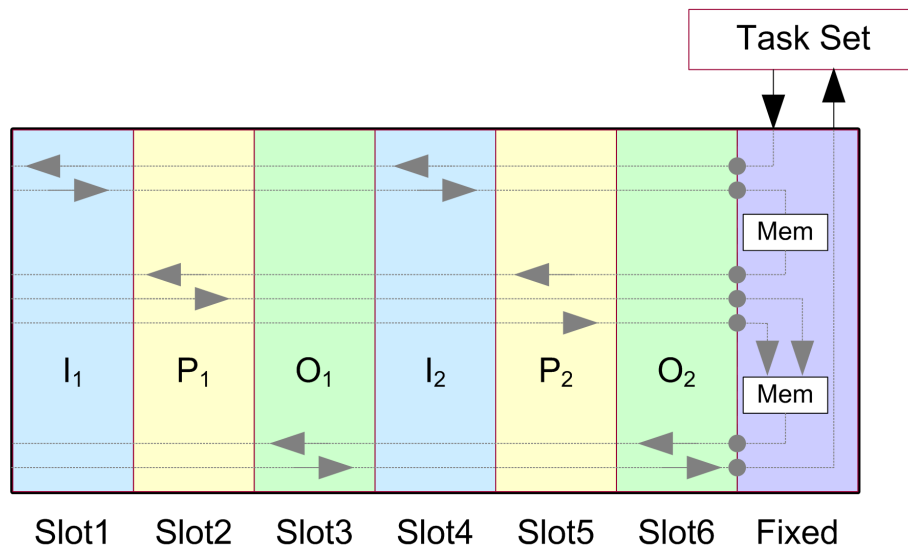


Abbildung 2.1: FPGA mit Slots

Somit ist es möglich, neue Funktionalitäten zur Laufzeit hinzuzufügen. Ist die neue Funktionalität komplett geladen, kann diese einfach hinzugeschaltet werden. Ein weiterer Vorteil ist die Möglichkeit, vorhandene Funktionalitäten zu verdrängen. Soll eine Anwendung verdrängt werden, werden die Bereiche, die von der auszutauschenden Task benutzt werden, deaktiviert. Dann wird die neue Funktionalität geladen und die Bereiche wieder aktiviert, so dass die neue Funktionalität genutzt werden kann.

In Abbildung 2.1 ist ein partiell rekonfigurierbares FPGA mit einer geladene Task abgebildet. Der rechte Slot ist eine nicht austauschbare Komponente (als 'fixed' gekennzeichnet). Über diesen Slot, der die zentralen und nicht austauschbaren Teile eines IFBs beinhaltet, wird die Verbindung zu externen Anwendungen hergestellt. Zusätzlich fungiert dieser Slot als Speicher. Dieser Speicher steht den Anwendungen in allen Slots zur Verfügung, da der Inhalt dieses Slots niemals ausgetauscht (also rekonfiguriert) wird.

Der fest installierte Slot enthält im Falle des IFB neben dem Speicher im Wesentlichen eine Kontrolleinheit. Diese wird u.a. für die Steuerung und Überwachung der Rekonfigurierung benötigt. Denn bevor eine Rekonfigurierung durchgeführt werden kann, müssen einige Bedingungen erfüllt sein. Beispielsweise muss sichergestellt sein, dass zum Zeitpunkt der Rekonfigurierung keine zu rekonfigurierende Komponente aktiv ist. Details zu der Kontrolleinheit werden in Kapitel 4 genauer erläutert.

2.2 Interface Block (IFB)

Im Folgenden wird die eigentliche Anwendung, die auf dem FPGA implementiert werden soll, näher erläutert. Dazu siehe auch [Ihm03] und [Ihm02].

Ein *Interface Block* (IFB) fungiert, wie bereits in der Einleitung beschrieben, als "Übersetzungskomponente". Existieren zwei Kommunikationspartner A und B (im Folgenden als Task bezeichnet), deren Kommunikationsstruktur (Protokolle) inkompatibel sind, so ist generell eine direkte Kommunikation zwischen diesen beiden Tasks nicht möglich. Der IFB ist in der Lage, mit beiden Komponenten zu kommunizieren und die Daten, die zwischen A und B ausgetauscht werden, so zu transformieren, dass die empfangende Task die Daten vom sendenden Task verstehen kann. Der IFB ist dabei für A und B transparent.

Wie in Abbildung 2.2 dargestellt, besteht der IFB generell aus drei Hauptkomponenten: dem *Protocol Handler*(PH), dem *Sequence Handler*(SH) und der *Control Unit*(CU). Der Aufbau und die Funktionsweise dieser Komponenten werden im Folgenden erklärt.

Der **Protocol Handler** ist die Schnittstelle zwischen den externen Tasks und den anderen Komponenten des IFB. Der PH besteht aus einzelnen PH_{Modes} , die mit jeweils der Schnittstelle einer Task verbunden sind und die Kommunikation mit dieser Task realisieren. Um die Daten korrekt zu empfangen, muss der PH_{Mode} das Protokoll der externen Task verarbeiten können. Dazu besteht der Mode aus einer *Finite State Machine* (FSM), die das komplementäre Protokoll dieser Task implementiert. Dabei werden aus dem Protokoll, das von der Task empfangen wird, Kontrollbits ausgefiltert, die einen korrekten Kommunikationsablauf sicherstellen sollen. Ein PH_{Mode} kann ausschließlich mit der Task kommunizieren, dessen komplementäres Protokoll implementiert wird. Es

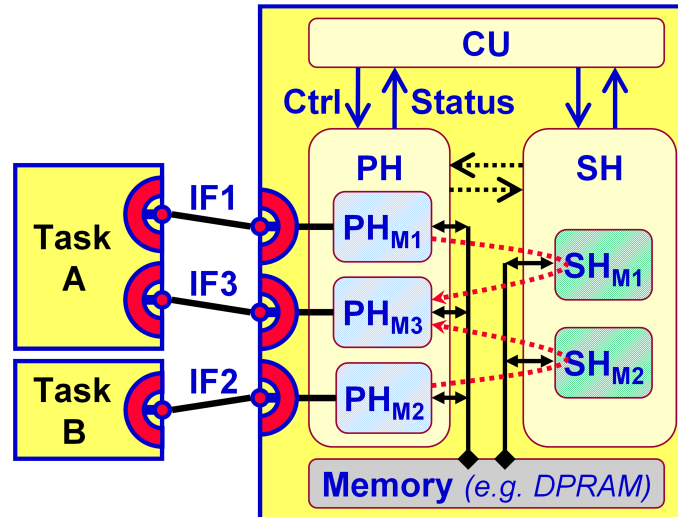


Abbildung 2.2: IFB Makrostruktur

wird also für jede Task, die mit dem IFB verbunden sind, ein PH_{Mode} benötigt. Die FSM innerhalb des Modes benötigt noch zusätzliche Zustände, um die Interaktion mit den anderen Komponenten des IFBs zu realisieren.

Der **Sequence Handler** ist die Komponente, die die "Übersetzung" der Daten vornimmt. Er besteht aus einer Anzahl von SH_{Modes} und zwei Speichern (Data Reader und Data Writer). Die SH_{Modes} bestehen, wie die PH_{Modes} , aus Automaten und modifizieren die Daten. Das sogenannte *IFD-Mapping* (Kapitel 2.4) gibt dabei an, wie die Daten aus dem Data Reader auf die Daten im Data Writer abzubilden sind. Die beiden Speicher sind über Datenbusse mit den PH_{Modes} und den SH_{Modes} verbunden und dienen als Zwischenspeicher für eingehende bzw. ausgehende Datenpakete. Die Größe des Speichers entspricht genau der Menge an Daten, die empfangen bzw. durch die Transformation generiert und nachfolgend versendet werden. Jedes Bit hat dabei eine genau festgelegte Position in dem Speicher, der speziell für dieses Szenario synthetisiert wurde. Liest ein PH_{Mode} ein Bit ein, so wird die vorhandene Information an der Stelle dieses Bits überschrieben. Die Synthese des Speichers erfolgt aufgrund der Informationen, die in dem IFD-Mapping enthalten sind.

SH und PH werden durch die **Control Unit** koordiniert. Die CU ist bspw. für die Arbitrierung der Datenbusse zuständig. Da allgemein davon ausgegangen wird, dass ein IFB mit mehreren sendenden und mehreren empfangenden Tasks verbunden ist (Multi-Task IFB), müssen die Zugriffe zeitlich gesteuert werden. Dazu existieren die $CTRL_{In}$ Komponente und die $CTRL_{Out}$ Komponente (beide im Folgenden als *Scheduler* bezeichnet). Erstere vergibt den Datenbus für eingehende Daten, letztere den Datenbus für ausgehende Daten. Somit wird sichergestellt, dass immer nur ein PH_{Mode} Daten in den Data Reader schreibt und ein PH_{Mode} Daten aus dem Data Writer liest. Des weiteren steuert die CU die Sequence Handler Modes. Erst wenn die Modes ein entsprechendes Signal von der CU erhalten, beginnen sie mit der Verarbeitung (Mapping) der Daten. Dieses Signal wird vom *Scoreboard* gesendet. Erst wenn durch das Scoreboard entsprechende

Signale gesendet wurden, kann ein Scheduler einen Datenbus freigeben oder ein SH_{Mode} mit der Transformation der Daten beginnen. In Kapitel 2.5 wird das Scoreboard detaillierter erklärt, da es eine wichtige Komponente zur Optimierung darstellt. Die letzte Komponente in der CU ist die *Reconfiguration Unit*. Sie überwacht die Rekonfigurierung und ist eine zentrale Komponente bei der Flächenoptimierung (Kapitel 4).

Die Kommunikation zwischen diesen Komponenten erfolgt über ein *fully interlocked protocol*. Die Funktionsweise der Komponenten wird im folgenden Abschnitt anhand eines Beispiels verdeutlicht

2.3 Datenfluss im IFB

Es wird von zwei Tasks A und B ausgegangen. Die Tasks besitzen inkompatible Protokolle, müssen aber Daten austauschen. Zwischen beiden Tasks wird ein IFB eingesetzt und mit den beiden Tasks verbunden. Sendet nun Task A Daten, werden sie vom IFB empfangen.

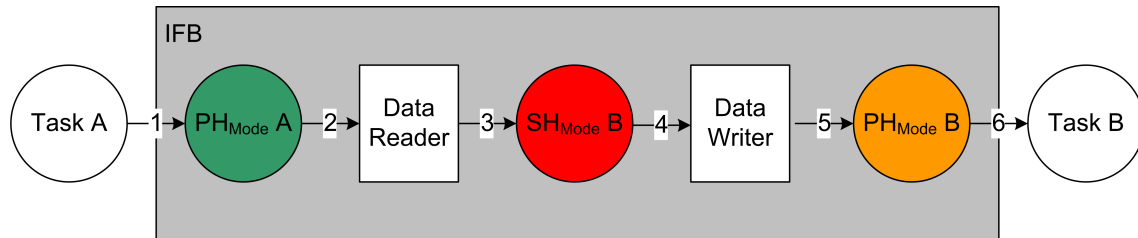


Abbildung 2.3: IFB Datenfluss

Schritt 1 Der PH_{Mode} A bewirbt sich um den Zugriff auf den Datenbus zum Data Reader. Nachdem der Scheduler den Bus zugeteilt hat, werden die Daten vom PH_{Mode} gelesen.

Schritt 2 Der PH_{Mode} leitet die Nutzdaten zum Data Reader, die dort gespeichert werden.

Schritt 3 Wurden die Daten komplett eingelesen, startet das Scoreboard den entsprechenden SH_{Mode} , der die Daten aus dem Data Reader liest und modifiziert.

Schritt 4 Die transformierten Daten werden in den Data Writer geschrieben.

Schritt 5 Hat der SH_{Mode} B seine Arbeit beendet, kann sich der PH_{Mode} B erfolgreich um den Datenbus zum Data Writer bewerben. Darf der PH_{Mode} den Datenbus belegen, liest er die Daten aus dem Data Writer, integriert diese in das ausgehende Protokoll.

Schritt 6 Darauf sendet der PH_{Mode} B die Daten an Task B.

2.4 IFD-Mapping

Das IFD-Mapping (*Interface Definition Mapping*) beinhaltet eine Menge von Regeln, die beschreiben, wie eingehende auf ausgehende Daten abgebildet werden. Eine solche Regel wird *Mapping Equation* (MapEqn) genannt und hat die Form

$$MapEqn : data_{Out} \leq f_{Map}(data_{In,1}, \dots, data_{In,k});$$

Ein ausgehendes Datenpaket kann also von einem oder mehreren eingehenden Datenpaketen abhängig sein. Die Umwandlung der Daten aus den Datenpaketen erfolgt über vier mögliche Grundoperationen:

- Zuweisung konstanter Werte
- Zuweisung eingehender Bits (ohne diese zu verändern)
- Anwenden von booleschen Funktionen auf eingehende Bits
- Verwenden eines endlichen Zustandsautomaten (FSM) zum Erzeugen von Bits

Bevor hier ein kurzes Beispiel zur Verdeutlichung gegeben wird, wird erklärt, wie die Daten einer Task empfangen und vorbereitend verarbeitet werden, bevor sie abgebildet werden können.

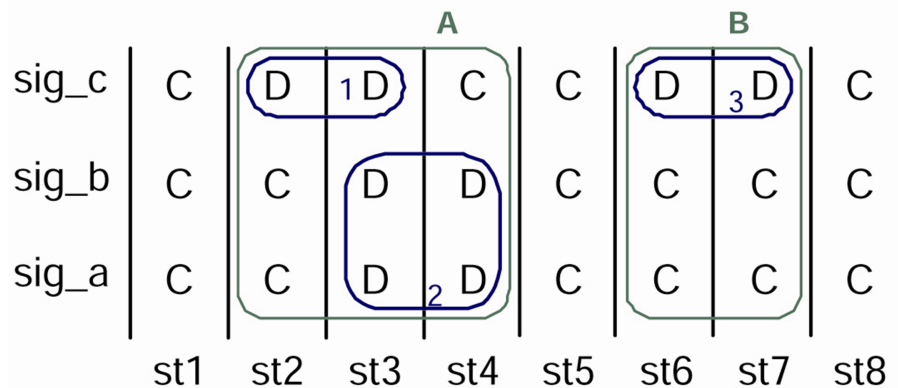


Abbildung 2.4: Ein Basic Block aus einem Protokoll mit Kontroll- und Nutzdaten

Die Daten einer Task sind in einem Protokoll (siehe Abbildung 2.4) enthalten. Dieses Protokoll wird in drei Schritten zergliedert.

Der erste Schritt analysiert das Protokoll und zerteilt es in **Basic Blocks**. Basic Blocks sind Teile des Protokolls und bestehen aus einer Menge von Zuständen und Transitionen. Innerhalb dieses Basic Blocks darf es keine Verzweigung geben, d.h. es ist eine sequentielle Folge von Zuständen. Ausführliche Informationen zu Basic Blocks und deren Eigenschaften sind in [Lok05] zu finden.

Abbildung 2.4 ist die Darstellung eines Basic Blocks als Matrix. Diese Matrix besteht aus Zuständen (Spalten) und den Signalen (Zeilen). In den einzelnen Zellen der Matrix sind die Bits eines Protokolls als Redundanz oder Informationen vorhanden. Die Redundanz liegt in Form von Kontrollbits (**C**) vor. Die Informationen enthalten die eigentlichen Datenbits (**D**). Um das Einlesen bzw. Abspeichern der Daten im Data Reader zu vereinfachen, werden zusammenhängende Bits (also Daten, die sich in angrenzenden Zellen befinden) zu Datenpaketen zusammengefasst. Hierbei werden zuerst die Zeilen nach zusammenhängenden Bits durchlaufen, dann erst die Spalten (deswegen ergibt sich im obigen Beispiel das Datenpaket 1 in der oberen Zeile und das Datenpaket 2 in den beiden darunterliegenden). Im nächsten Schritt werden sogenannte *Frames* gebildet. Frames enthalten mindestens ein Datenpaket und haben eine eindeutige ID. Überlappen sich zwei Datenpakete im gleichen Zustand, werden diese zu einem einzigen Frame zusammengefasst. Im obigen Beispiel überlappen sich die Datenpakete 1 und 2 und ergeben den Frame A, Datenpaket 3 hat keine Überschneidung mit anderen Datenpaketen und bildet so den Frame B. Frames kapseln die Daten, die ohne Unterbrechung zwischen PH und SH übertragen werden. Für diese Übertragung muss der entsprechende Datenbus im IFB zugeteilt werden. Im Folgenden soll in dieser Arbeit davon ausgegangen werden, dass jeder Frame immer exakt ein Datenpaket beinhalte.

Das folgende Beispiel beinhaltet drei der vier Grundoperationen des Mappings und soll die Funktionsweise des IFD-Mappings verdeutlichen:

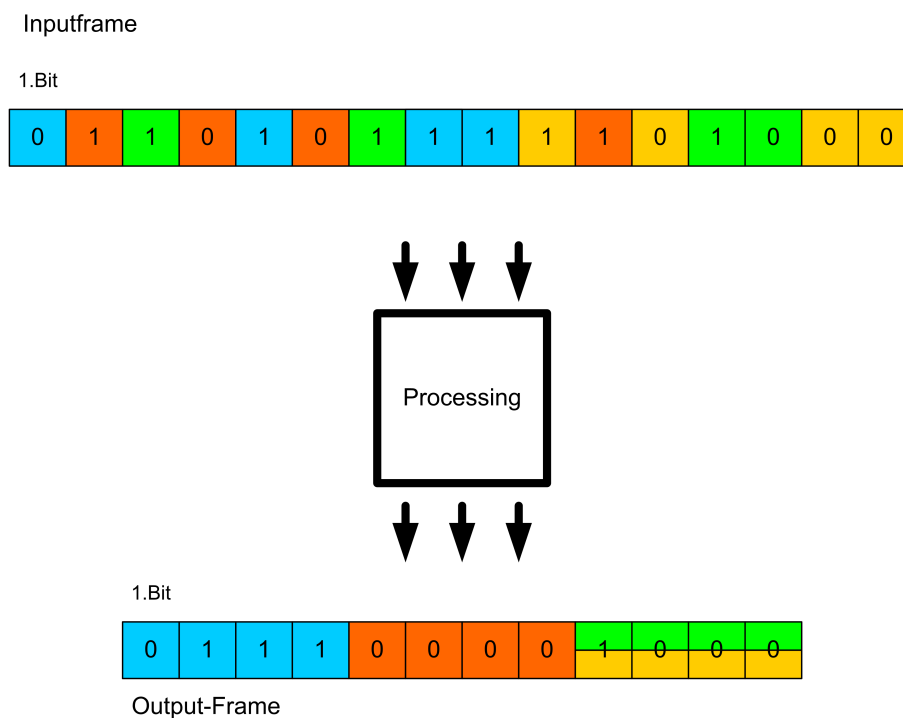


Abbildung 2.5: Funktion des IFD-Mappings

Die einzelnen Bits in Abbildung 2.5 wurden farblich markiert, um zu verdeutlichen, was während des Mappings geschieht. Die hellblau gefärbten Bits werden auf die ersten vier Bits des Output-Frames abgebildet. Der Inhalt der nächsten vier Bits wird durch eine '0' überschrieben und auf die mittleren vier Bitpositionen abgebildet. Die gelben und grünen Bits wurden bitweise '&' verknüpft und die Ergebnisse auf die letzte vier Bits abgebildet.

Die entsprechenden MappingEquations zu diesem Beispiel sehen folgendermaßen aus. Dabei steht 'P' für ein Datenpaket.

- $P_{Out}[1:1] \leq P_{In}[1:1];$
- $P_{Out}[2:2] \leq P_{In}[5:5];$
- $P_{Out}[3:4] \leq P_{In}[8:9];$
- $P_{Out}[5:8] \leq "0000";$
- $P_{Out}[9:9] \leq (P_{In}[3:3] \& P_{In}[10:10])$
- $P_{Out}[10:10] \leq (P_{In}[7:7] \& P_{In}[12:12])$
- $P_{Out}[11:11] \leq (P_{In}[13:13] \& P_{In}[15:15])$
- $P_{Out}[12:12] \leq (P_{In}[14:14] \& P_{In}[16:16])$

Aus diesen Mapping Equations kann also die Information extrahiert werden, welches Bit aus dem Input-Frame für welches Bit des Output-Frames benötigt wird. Einzelne Bits können dabei mehrfach genutzt werden.

2.5 Ausführung als Pipeline

Eine weitere grundlegende Eigenschaft des IFBs besteht darin, dass man die Arbeitsschritte aus Kapitel 2.3 in *Stufen* zusammenfassen kann. In der ersten Stufe werden Daten eingelesen, in der zweiten werden diese verarbeitet und in der dritten und letzten Stufe werden sie dann versendet. Diese einzelnen Stufen können zu einer Pipeline zusammengefasst werden und ergeben dann das so genannte *I-P-O Schema*: Input-Processing-Output. Wie bei einem Fließband gilt, dass eine Stufe erst dann begonnen werden kann, wenn die vorige Stufe beendet ist. Für die Einhaltung des I-P-O Schemas im IFB ist das *Scoreboard* zuständig. Es kann anhand von Bedingungen feststellen, ob und welche Stufe als nächstes begonnen wird. Diese Bedingungen werden im Folgenden aufgeführt und erklärt [Ave05]:

Eingabe Diese Bedingungen beziehen sich auf die Daten, die von sendenden Tasks empfangen werden.

- Ein Frame enthält n Datenpakete.
- Ein Ausgangsdatenpaket kann nicht als Eingangsdatenpaket genutzt werden.
- Wurden alle Daten im Data Reader von einem SH_{Mode} gelesen und verarbeitet, kann ein neuer Frame gelesen werden.
- Nachdem der PH_{Mode} ein Frame vollständig eingelesen hat, wird dies dem Scoreboard gemeldet.

Verarbeitung Diese Bedingungen beziehen sich auf den Start der Verarbeitung. Diese darf beginnen, wenn alle Daten vorhanden sind.

- Sobald alle für einen Sequence Handler Mode notwendigen Datenpakete vollständig im Data Reader sind und alle entsprechenden Daten im Data Writer gesendet wurden, startet das Scoreboard den entsprechenden SH_{Mode} .

Ausgabe Hier prüft das Scoreboard, ob Daten versendet werden dürfen.

- Ein Frame kann ausgegeben werden, wenn der SH_{Mode} beendet ist und alle abgebildeten Daten im Data Writer abgespeichert wurden.

Das Scoreboard ist damit eine zentrale Komponente, die die Befolgung des I-P-O Schemas und damit die Kausalität des Datenverkehrs kontrolliert. Es wertet die Kontrollsignale der beteiligten Komponenten aus und verhindert oder lässt zu, dass ein PH_{Mode} oder SH_{Mode} mit seiner Ausführung fortfahren kann. Es existieren allerdings Szenarien, in denen das I-P-O Schema *nicht* eingehalten werden kann. Um die Problematik und deren Lösung zu verdeutlichen, wird zunächst der allgemeine Aufbau eines PH_{Modes} gezeigt.

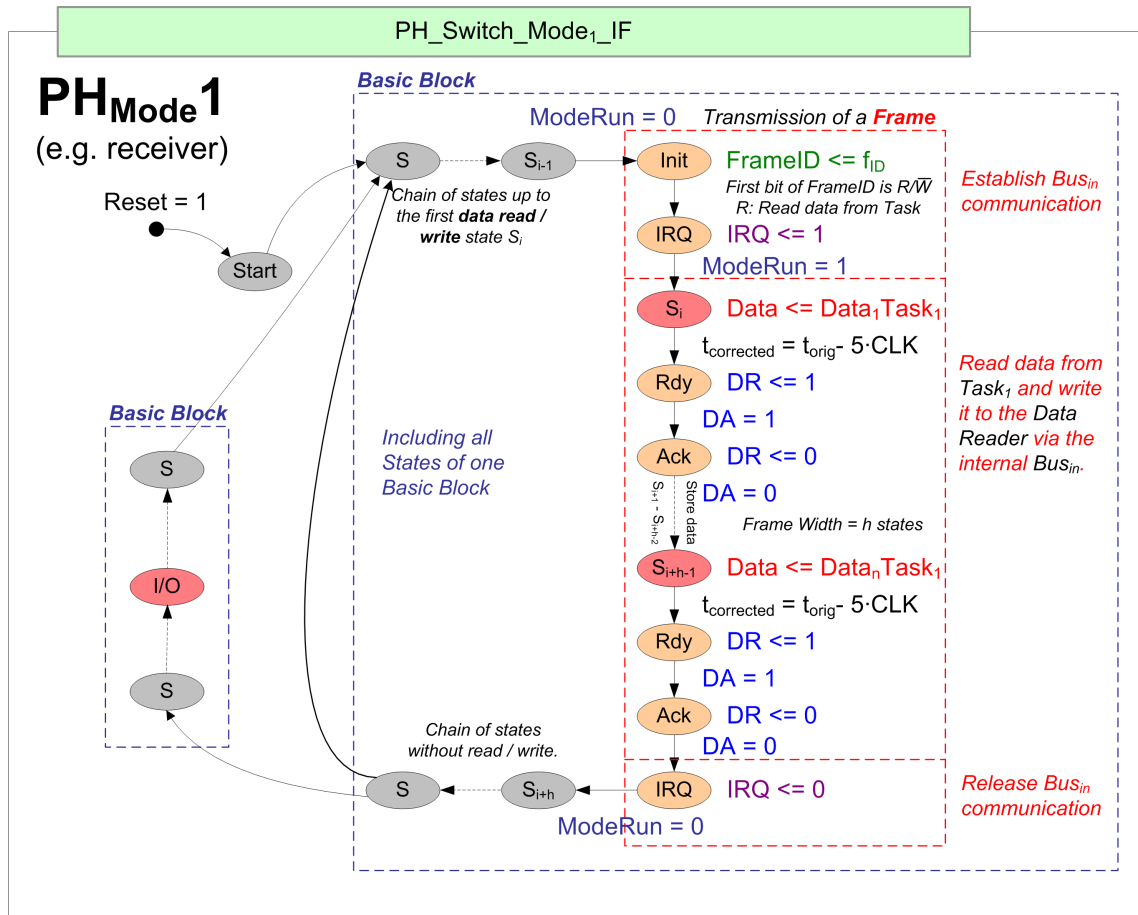
Abbildung 2.6: Aufbau eines PH_{Modes}

Abbildung 2.6 zeigt einen beispielhaften PH_{Mode} , dessen Automat einen **Start** Zustand hat, aus dem in den eigentlichen Lesevorgang gewechselt wird. In *Establish Bus_{In} communication* fordert der PH_{Mode} den Buszugriff an. Mit einem einmaligen Durchlauf kann der Input-Frame des blau umrandeten Basic Blocks genau einmal gelesen werden. Nach dem Lesen wird der Datenbus in dem Zustand *Release Bus_{In} communication* wieder frei gegeben.

Problematisch wäre nun ein Szenario, in dem ein Input-Frame zweimal gelesen werden muss, um alle benötigten Daten für einen Output-Frame einzulesen. Somit müsste der Basic Block, der den Input-Frame beinhaltet, zweimal ausgeführt werden.

Ebenso wäre ein Szenario denkbar, in dem für einen Input-Frame mehrere Instanzen eines Output-Frames notwendig wären. Das würde implizieren, dass auch das Processing mehrfach ausgeführt werden müsste. Der Scheduler im IFB lässt nur die einmalige Ausführung des Inputs, Processings und Outputs pro Kommunikationszyklus zu. Wenn also innerhalb einer Abbildungsvorschrift mehrere Instanzen eines Frames benötigt werden, erfordert dies eine Modifikation der PH_{Modes} .

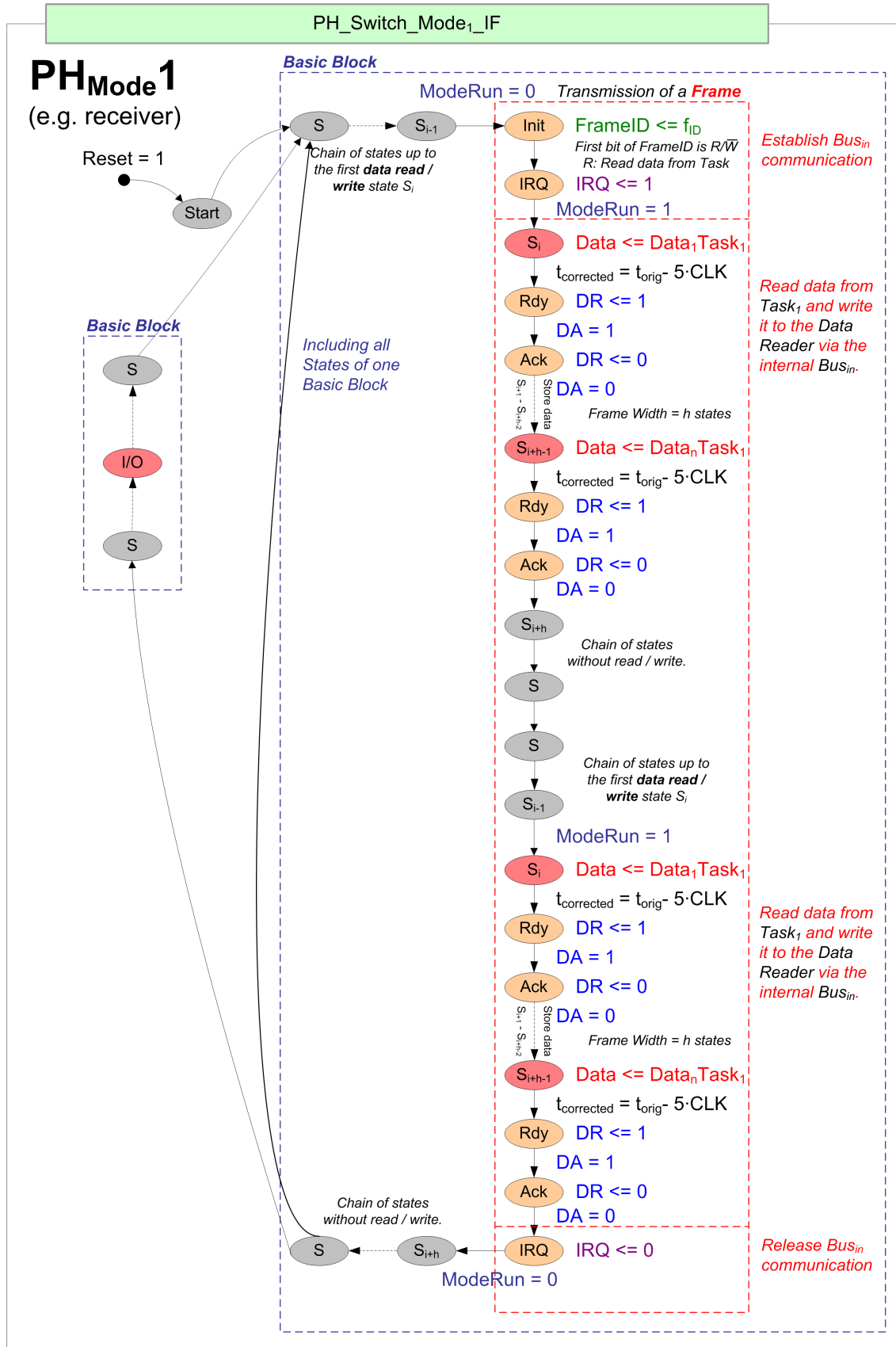


Abbildung 2.7: Ein modifizierter PH_{Modes}

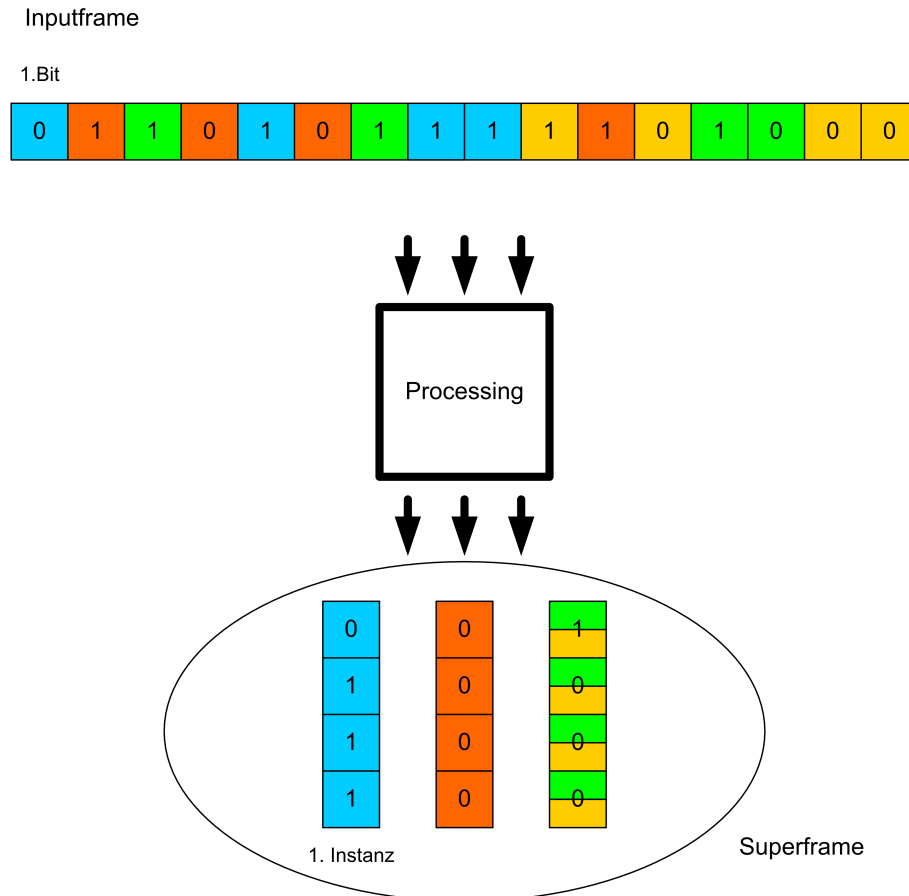


Abbildung 2.8: Nutzung von Instanzen und Superframes

In Abbildung 2.7 ist ein modifizierter PH_{Mode} zu sehen. Dieser PH_{Mode} besitzt einen Automaten, in dem ein Basic Block "verdoppelt" wurde. Das führt dazu, dass der ursprüngliche Basic Block *zweimal* direkt hintereinander ausgeführt wird. Die "Vervielfachung" eines Basic Blocks ist immer dann erlaubt, wenn ein Basic Block eine Selbsttransition besitzt. Der durch diese Transformation entstandene Frame wird als *Superframe* bezeichnet und erlaubt es, mehrere Instanzen eines Pakets zu lesen bzw. zu schreiben.

Abbildung 2.8 zeigt ein Szenario, in dem durch die Verwendung von Superframes das I-P-O Schema eingehalten werden kann. Das Beispiel zeigt einen relativ großen Input-Frame, der in drei Instanzen des Output-Frames versendet werden soll. Hierbei wurde der Superframe durch die Verdreifachung des Basic Blocks gebildet.

2.6 Detaillierte Darstellung des IFB Datenflusses

Um die nachfolgende Optimierung der Latenz besser verstehen zu können, wird in diesem Abschnitt noch einmal ausführlich der Datenfluss des IFB betrachtet. Dazu werden die einzelnen Aktionen aufgeführt, um die Interaktion zwischen den einzelnen Komponenten inklusive des *fully interlocked protocol* aufzuzeigen.

Takt	Phase	Aktion	Komponente
1	1	SetFrameId	PH _{Mode}
2	1	SetIRQ	PH _{Mode}
3	1	GrantBus	CTRL _{In}
4	2	Daten lesen	PH _{Mode}
5	2	SetDR	PH _{Mode}
6	2	Daten in Speicher übernehmen	Data Reader
7	2	SetDA	Data Reader
8	2	ResetDR	PH _{Mode}
9	2	ResetDA	Data Reader
10	3	ResetIRQ	PH _{Mode}
11	3	AssignFrame or ReleaseBus	CTRL _{In}
12	3	Frame wird in P _{in} Register übernommen	CU
13	4	ModifyData	SH _{Mode}
14	4	Set SH _{Mode} Complete	SH _{Mode}
15	4	Set SH _{Mode} Complete	CU
16	5	SetFrameId	PH _{Mode}
17	5	SetIRQ	PH _{Mode}
18	5	GrantBus	CTRL _{Out}
19	6	SetDR	PH _{Mode}
20	6	Daten auf Datenbus	DataWriter
21	6	SetDA	Data Writer
22	6	Daten schreiben	PH _{Mode}
23	6	ResetDR	PH _{Mode}
24	6	ResetDA	Data Writer
25	7	ResetIRQ	PH _{Mode}
26	7	AssignFrame or ReleaseBus	CU
27	7	Reset SH _{Mode} Complete	Scoreboard

Tabelle 2.1: Detaillierte Darstellung des Datenflusses im IFB

Die Tabelle 2.1 beschreibt Taktzyklen-genau das Einlesen, das Verarbeiten und das Senden eines Frames mit jeweils einem Bit. Dabei können sieben Phasen unterschieden werden. Die 27 Schritte wurden direkt aus den kommunizierenden Automaten des IFBs abgeleitet und beschreiben die präzise Übertragung eines Frames. Die Schritte stellen eine Verfeinerung des in Kapitel 2.3 angegebenen Datenflusses dar.

- Phase 1 – Establish** Die erste Phase (Takt eins bis drei) beinhaltet das Festlegen der Frame ID und die Bewerbung um den Datenbus.
- Phase 2 – Read** In der zweiten Phase (Takt vier bis neun) läuft der eigentliche Lesevorgang der Daten ab. Diese Phase vervielfacht sich je nach Anzahl der einzulesenden Bits.
- Phase 3 – Release** Das Einlesen wurde abgeschlossen und der Datenbus wird wieder freigegeben (Takt zehn bis zwölf).
- Phase 4 – Process** In den Takten 13 bis 15 werden die Daten modifiziert und abgespeichert.
- Phase 5 – Establish** Nun sollen die Daten an die empfangsbereite Task gesendet werden. Dazu wird der Datenbus benötigt. Dazu werden (wie beim Einlesen) drei Takte benötigt.
- Phase 6 – Write** Takte 19 bis 24 beschreiben das Schreiben eines Bits zu einer empfangsbereiten Task. Werden mehrere Bits geschrieben, müssen diese sechs Takte für jedes Bit wiederholt werden.
- Phase 7 – Release** Das Senden der Daten ist abgeschlossen und die letzten drei Takte geben u.a. den Datenbus wieder frei.

Die ersten drei Phasen **Establish**, **Read** und **Release** sind Teil jeder Input Stage, die Phase **Process** steht für die gleichnamige Stage im I-P-O Schema. Die letzten drei Phasen werden der Output Stage zugeordnet.

3 Datenflussoptimierung

In diesem Kapitel wird erläutert, wie eine Optimierung des IFB-internen Datenflusses erreicht werden kann. Dazu wird zunächst die nicht optimierte Ausführung des IFB vorgestellt. Anschließend werden zwei Verfahren zur Verkürzung der Latenz beschrieben und die erzielten Ergebnisse miteinander verglichen.

3.1 Ausführung des IFB als Pipeline

In diesem Abschnitt wird zunächst das ursprüngliche Ausführungsmodell des IFBs ohne Optimierung beschrieben. Dieses Modell dient als Basis für die beiden in diesem Kapitel vorgestellten Optimierungsverfahren. In Kapitel 2.5 wurde bereits die Ausführung eines Kommunikationszyklus in den Schritten Input-Processing-Output vorgestellt. Dort wurde bereits angesprochen, dass diese Schritte als Stages einer Pipeline angesehen werden können, deren Organisation durch das Scoreboard sichergestellt wird. Da Protokolle zyklisch ausgeführt werden, wiederholen sich die Stages Input, Processing und Output bei jeder Ausführung eines Kommunikationszyklus. Als Teil des Scheduling repräsentieren die Input und Output Stages Frames, die vom Protocol Handler empfangen bzw. versendet werden. Entsprechend existieren Processing Stages für jede Datentransformation (Mapping Equation) im Sequence Handler.

In Abbildung 3.1 wird das Scheduling für das einfachste Szenario einer Kommunikation bestehend aus einem Input-Frame, der Transformation und einem Output-Frame dargestellt. Gegeben sind die ersten beiden Zyklen; alle weiteren Zyklen können entsprechend konstruiert werden. Dieses Scheduling wird im Folgenden als simple Schedule bezeichnet.

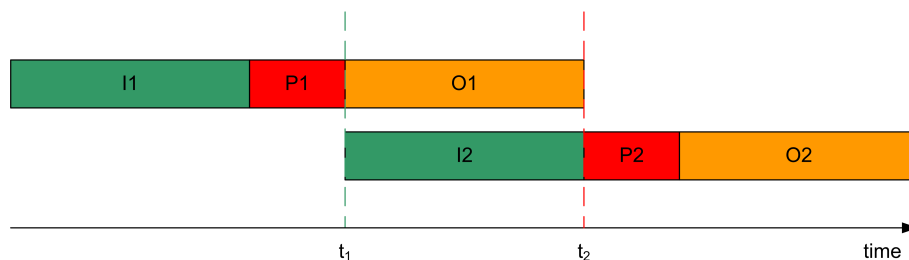


Abbildung 3.1: simple Schedule I

Bevor eine Stage ausgeführt werden kann, müssen bestimmte Bedingungen erfüllt sein, wie sie in Kapitel 2.5 informal angegeben wurden. Dabei existieren je nach Schedulingverfahren charakteristische Bedingungen für die Input, Processing und Output-Stages:

Input Bedingung

Die Input Stage eines Kommunikationszyklus darf erst dann ausgeführt werden, wenn die Processing Stage des vorigen Kommunikationszyklus beendet ist (hier durch die grüne Linie zum Zeitpunkt t_1 gekennzeichnet), denn erst nachdem alle Daten aus dem Data Reader verarbeitet wurden, dürfen sie durch neue Daten überschrieben werden.

- $\text{Input}(x)^1$ nach $\text{Process}(x-1)$

Processing Bedingungen

Die Processing Stage darf erst dann beginnen, wenn alle Daten der Input Stage eingelesen und im Data Reader gespeichert sind (t_2 , rote Linie). Diese Bedingung wird durch das I-P-O Schema vorgegeben und vom Scoreboard kontrolliert. Außerdem muss im vorigen Kommunikationszyklus das Versenden der Daten abgeschlossen sein (in diesem Beispiel ebenfalls zum Zeitpunkt t_2). Würde die Processing Stage früher beginnen, könnten noch nicht versendete Daten aus dem vorigen Zyklus durch neu berechnete Daten des aktuellen Kommunikationszyklus überschrieben werden.

- $\text{Process}(x)$ nach $\text{Input}(x)$
- $\text{Process}(x)$ nach $\text{Output}(x-1)$

Output Bedingung

Die Output Stage muss auf die Beendigung der Processing Stage warten, da erst dann alle Daten verarbeitet sind. Diese Bedingung ist ebenfalls durch die Kausalität im I-P-O Schema vorgegeben.

- $\text{Output}(x)$ nach $\text{Process}(x)$

In dem beschriebenen Beispiel werden genau so viele Daten eingelesen wie versendet werden. In Abbildung 3.2 ist ein modifiziertes Beispiel angegeben, in dem der Input-Frame kleiner als der Output-Frame ist. Die Input Stage zum Zeitpunkt t_2 beendet, aber die Output Stage des vorigen Zyklus erst zum Zeitpunkt t_3 . Die Processing Stage kann also erst zum Zeitpunkt t_3 gestartet werden.

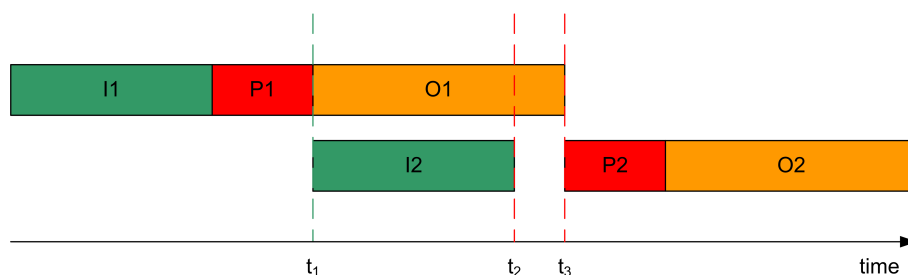


Abbildung 3.2: simple Schedule II

¹das x bezeichnet hier die Nummer des Kommunikationszyklus

Das Scheduling in den Abbildungen 3.1 und 3.2 vorgestellten Kommunikationszyklen lässt sich nicht weiter optimieren, wenn man davon ausgeht, dass die repräsentierten Frames keine Superframes sind. Das bedeutet, dass innerhalb der Input und Output Stage jeweils genau eine Instanz der ursprünglichen Frames enthalten ist.

3.2 Optimierungskonzept

Die Datenflussoptimierung zur Reduzierung der Latenz nutzt eine Besonderheit des I-P-O Schemas aus. Eine Optimierung der Latenz des IFB ist möglich, wenn bedingt durch das IFD-Mapping ein Superframe für die Output Stage und damit auch für die Processing Stage vorliegt. Wie in Kapitel 2.4 beschrieben, werden in einem solchen Fall eine große Menge eingehender Daten eines einzigen Frames auf mehrere Instanzen kleinerer ausgehender Frames abgebildet, die zu einem Superframe zusammengefasst werden. In Abbildung 3.3 ist ein solches Szenario dargestellt.

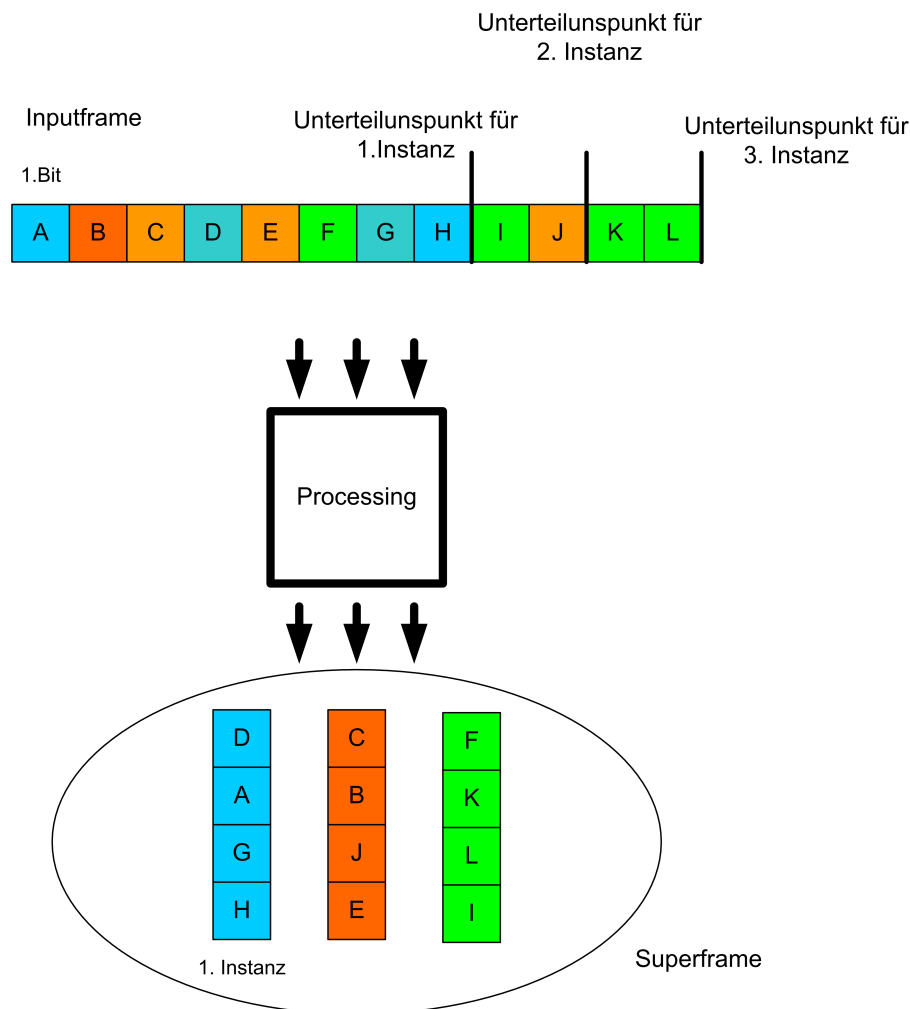


Abbildung 3.3: Einführung von Unterteilungspunkten innerhalb des Input-Frames

Das Beispiel in Abbildung 3.3 zeigt, dass die Möglichkeit besteht, dass die Daten der ersten Instanz im Output-Superframe bereits vollständig eingelesen wurden (beim ersten Unterteilungspunkt), bevor das letzte Bit des Input-Frames im IFB eintrifft.

Es werden in diesem Beispiel 16 Bits sequentiell eingelesen und ein Superframe bestehend aus drei Instanzen mit je vier parallelen Bits versendet. Die erste Instanz benötigt die Bits A, D, G und H des Input-Frames. Der SH_{Mode} kann die Verarbeitung dieser vier Bits starten, sobald diese im Speicher abgelegt sind (beim ersten Unterteilungspunkt). Dann versendet der entsprechende PH_{Mode} die erste Instanz des Superframes. Sind die Bits B, C, E und J komplett eingelesen (zweiter Unterteilungspunkt), können diese verarbeitet und die zweite Instanz kann versendet werden. Ist der Input-Frame komplett eingelesen (letzter Unterteilungspunkt), sind auch die Informationen für die letzte Instanz vorhanden, so dass diese transformiert und dann in das ausgehende Protokoll integriert werden können.

Um festzustellen, wann alle benötigten Daten für eine Instanz im Output-Superframe vollständig eingelesen sind, wird das IFD-Mapping analysiert, das in Kapitel 2.4 vorgestellt wurde. Da das IFD-Mapping die Informationen beinhaltet, welche Bits aus dem Input-Frame auf welche Bitpositionen des Output-Superframes gemappt werden, kann daraus abgeleitet werden, wann frühestens mit dem Processing der Daten einer Instanz innerhalb des Output-Superframes begonnen werden kann. Aus diesen Informationen ergeben sich die Unterteilungspunkte.

Somit lässt sich für jede Instanz innerhalb des Output-Superframes ableiten, wann mit der Verarbeitung der eingehenden Daten begonnen werden kann. Die Datenflussoptimierung besteht also darin, einen "großen" eingehenden Frame in mehrere Abschnitte zu unterteilen und diese dann frühzeitig zur berechnen und auszugeben. Diese Abschnitte werden im Folgenden als *Subframes* bezeichnet. Diese Subframes stellen einen genau spezifizierten Teil eines Frames dar, d.h. jeder Subframe ist genau einem Frame zugeordnet und beinhaltet einen Abschnitt zwischen zwei Unterteilungspunkten. Da ein Subframe bei einem Unterteilungspunkt beginnt und beim nächsten aufhört, ist es nicht möglich, dass Subframes sich überlappen.

$$\cap \text{Subframes} = \emptyset$$

Damit ergibt sich als alternative Beschreibung für einen Frame

$$\text{Frame} = \cup \text{Subframes}.$$

Allerdings müssen zwei Bedingungen eingehalten werden, damit die Unterteilung des Frames in Subframes erfolgen kann:

- Der Input-Frame beinhaltet Informationen für mehr als einen Output-Frame, so dass mehrere Instanzen benötigt werden. Besteht die Ausgabe dagegen aus nur einem Frame, müssen alle Daten eingelesen werden und es ergeben sich keine Unterteilungspunkte.
- Der Input-Frame muss eine serielle Folge von Datenbits beinhalten. Ist diese Bedingung nicht erfüllt, d.h. sind alle Eingangsdaten parallel, werden sie gleichzeitig eingelesen. Eine Unterteilung ist somit nicht möglich.

Sind diese Bedingungen erfüllt, können anhand der Unterteilungspunkte die Subframes definiert werden: Der erste Input-Subframe beginnt am Anfang des Input-Frames und reicht bis zum ersten Unterteilungspunkt (siehe Abbildung 3.3). Daran schließt sich der zweite Subframe bis zum nächsten Unterteilungspunkt an, daran wiederum der dritte usw.

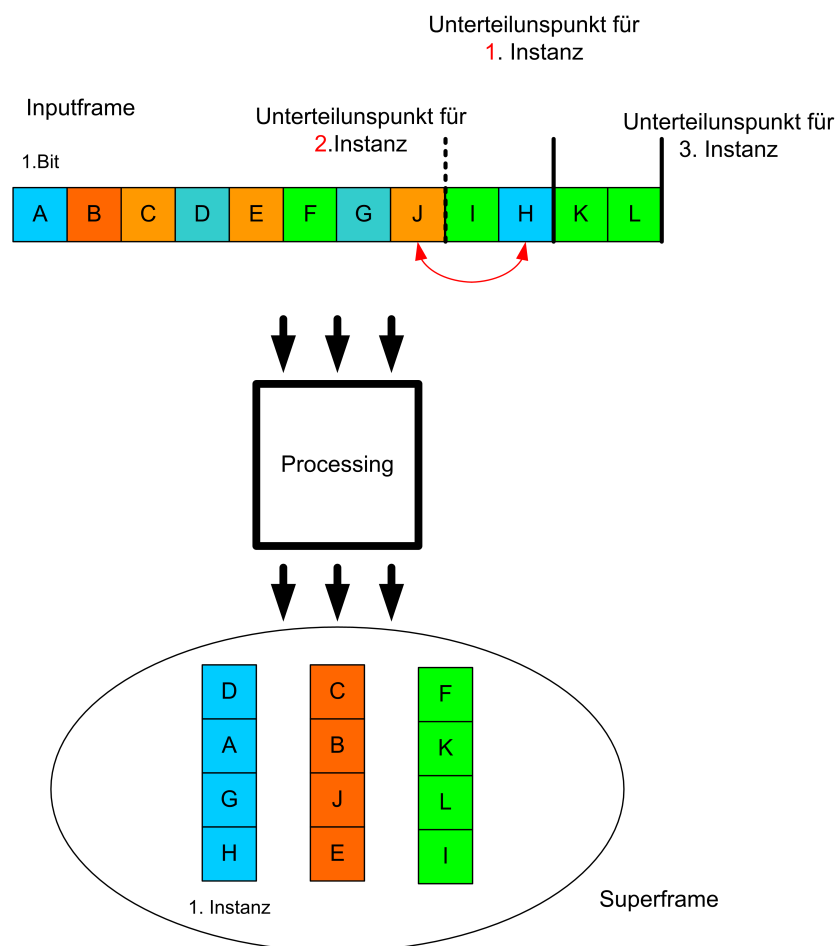


Abbildung 3.4: Beispiel für den Wegfall von Unterteilungspunkten

Dabei kann es zu folgender Situation kommen: Abbildung 3.4 ist eine Modifikation der Abbildung 3.3. In diesem Beispiel sind die Daten für die zweite Instanz bereits vor dem Unterteilungspunkt der ersten Instanz eingelesen. Die Reihenfolge, in der Subframes verarbeitet werden, ist aber fest vorgegeben. Dadurch entfällt der Unterteilungspunkt für die zweite Instanz. Erst nachdem der erste Unterteilungspunkt erreicht ist, kann die erste Instanz ausgeführt werden. Da die Daten für die zweite Instanz bereits vollständig vorliegen, können diese direkt im Anschluss an die erste Instanz transformiert werden.

Ein Nachteil dieses Optimierungsansatzes besteht im Folgenden: Benötigt eine Instanz Daten, die erst am Ende des Input-Frames eingelesen werden, so reduziert sich der Optimierungseffekt, da alle nachfolgenden Instanzen erst dann ausgeführt werden dürfen, wenn die Informationen für den Vorgänger komplett eingelesen und transformiert sind. Der schlimmste anzunehmende Fall ist der, dass die erste Output-Frame-Instanz das letzte einzulesende Bit benötigt. In diesem Fall ist der Optimierungseffekt gleich Null, da die Verarbeitung erst nach dem Einlesen des letzten Bits beginnt.

Um die Aufteilung der Input-Frames in der automatischen Generierung zu realisieren, existieren zwei Möglichkeiten. Die erste Möglichkeit beinhaltet das Einführen von Subframe IDs. Eine Subframe ID wird genau einem Subframe zugeordnet. Somit sind die Subframes eindeutig identifizierbar und unterscheidbar. Diese Verwendung der Subframe IDs kann dazu genutzt werden, um festzustellen, wann ein Subframe komplett eingelesen wurde. Wird das Ende eines Subframes beim Einlesen erreicht, wird die Subframe ID inkrementiert. Das ist das Signal, dass die Daten für die nächste Output-Frame-Instanz vorhanden sind und transformiert werden können.

Die zweite Möglichkeit besteht darin, dass der PH_{Mode} nach dem Einlesen eines Subframes den Datenbus wieder freigibt (**Release**) und sich dann für den neuen Subframe erneut darum bewirbt. Ein Frame wird somit nicht als Menge von Subframes verarbeitet, sondern als Menge von kleinen Frames angesehen. Dabei geht die eindeutige Zuordnung eines Subframes zu einem Frame nicht verloren!

3.3 Optimierungsverfahren

In den beiden folgenden Abschnitten werden zwei Optimierungsansätze für das Beispiel aus Abbildung 3.3 präsentiert. Die beiden Ansätze resultieren aus den beiden Möglichkeiten, die Subframes zu realisieren.

3.3.1 framebased schedule

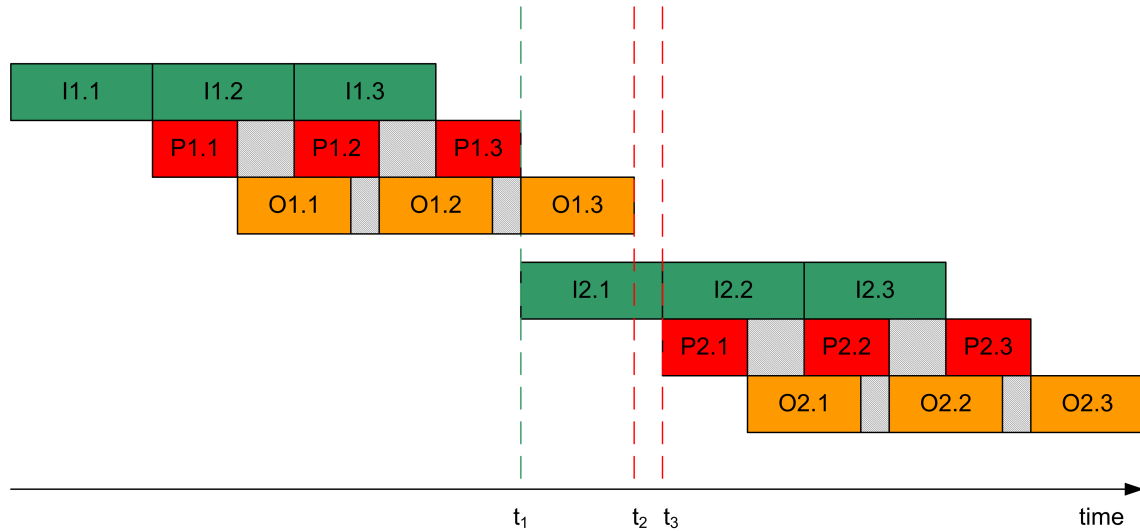


Abbildung 3.5: framebased Schedule

Dieses Verfahren verwendet die Subframe IDs. Wie bereits beschrieben, wird jedem Subframe eine solche ID zugewiesen, um die einzelnen Subframes voneinander zu unterscheiden. Kontrolliert und verwaltet werden die Subframe IDs vom PH_{Mode} , der an den Unterteilungspunkten die Subframe ID inkrementiert. Es ergibt sich folgende Struktur:

Wie in Abbildung 3.6 sichtbar, bleibt der Frame als solcher erhalten, es werden nur Abschnitte (die Subframes) in diesem Frame definiert.

Hat ein PH_{Mode} einen Subframe komplett eingelesen, muss der SH_{Mode} , der die Daten aus diesem Subframe transformiert, darüber informiert werden. Dazu sendet der PH_{Mode} über eine neue Leitung, die zwischen dem PH_{Mode} und dem SH_{Mode} eingerichtet wird, ein entsprechendes Signal an den SH_{Mode} . Diese zusätzliche Leitung fungiert als *Bypass* zur CU und ist notwendig, da das Scoreboard einen Start des SH_{Modes} erst dann zulassen würde, wenn der komplette Frame eingelesen ist. Zusätzlich benötigt der PH_{Mode} eine Erweiterung der Zustände, um die Subframe IDs zu verwalten und den Subframes zuzuweisen. Dabei kann die bereits vorhandene Frame ID erweitert werden, so dass sie auch die Subframe ID enthält. Die CU muss derart modifiziert werden, dass der SH_{Mode} bereits dann aktiviert wird, sobald der Datenbus vom PH_{Mode} arbitriert wurde, damit der SH_{Mode} rechtzeitig schon während des Einlesens mit der Verarbeitung beginnen kann.

Der SH_{Mode} benötigt zusätzliche Zustände, um die Subframe ID auszuwerten. Nach der Verarbeitung eines Subframes wartet der SH_{Mode} mit der weiteren Verarbeitung darauf, dass der PH_{Mode} die Subframe ID inkrementiert.

Input Bedingung

Wie im simple Schedule müssen auch bei diesem Optimierungsansatz bestimmte Bedingungen eingehalten werden, die durch die Linien in Abbildung 3.5 verdeutlicht werden.

Die erste Input Stage eines Kommunikationszyklus darf erst dann begonnen werden, wenn *alle* Verarbeitungsphasen aus dem vorigen Zyklus beendet wurden. In Abbildung 3.5 ist diese Bedingung zum Zeitpunkt t_1 erfüllt. Ein verfrühtes Einlesen könnte Daten im Data Reader überschreiben, die noch nicht von einem SH_{Mode} verarbeitet wurden.

- $\text{Input}(x.1)^2$ nach $\text{Process}(x-1.L)^3$

Processing Bedingungen

Eine Processing Stage darf – gemäß dem I-P-O Schema – erst dann ausgeführt werden, wenn die dazugehörige Input Stage beendet wurde. Bspw. muss P2.1 auf I2.1 warten (t_3). Analog zum simple Schedule gilt außerdem die Bedingung, dass die Output Stage des vorigen Zyklus beendet sein muss, um das Überschreiben nicht gesendeter Daten zu vermeiden (t_2). Beide Bedingungen sind in Abbildung 3.5 durch die roten Striche gekennzeichnet.

- $\text{Process}(x.1)$ nach $\text{Output}(x-1.L)$
- $\text{Process}(x.y)$ nach $\text{Input}(x.y)$

²die erste Ziffer beschreibt wieder die Nummer des Kommunikationszyklus, die zweite Ziffer die Nummer des Subframes

³das L kennzeichnet den letzten Subframe einer Stage, in diesem Fall wäre dies $\text{Process}(x-1.3)$

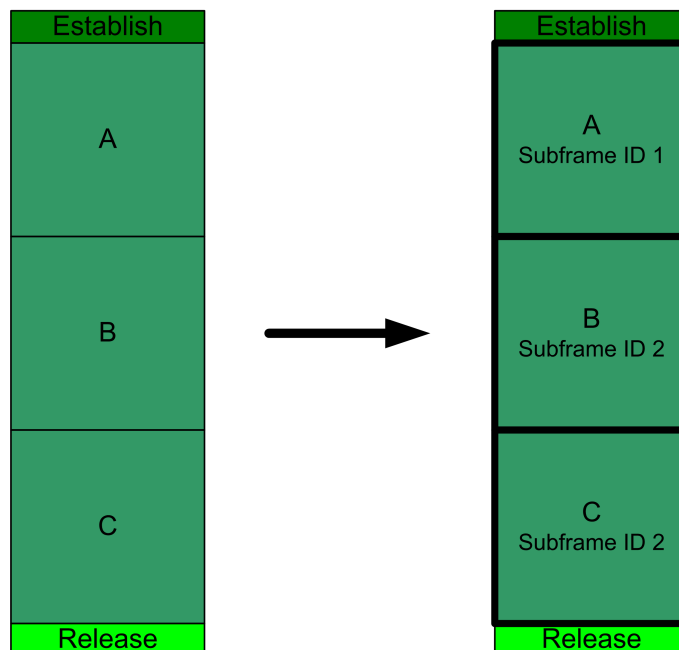


Abbildung 3.6: Links der ursprüngliche Frame, rechts der Frame mit der internen Aufteilung in die drei Subframes

Output Bedingung

Die Ausführung der Output Stage richtet sich wiederum nach dem Ende der zugehörigen Processing Stage. O2.1 darf nicht gestartet werden, bevor P2.1 beendet wurde. Diese Bedingung wird durch das Scoreboard kontrolliert, da sie durch die Kausalität im I-P-O Schema vorgegeben ist.

- Output(x.y) nach Process(x.y)

Somit kann eine Verbesserung erreicht werden. Die Verarbeitung in einem Kommunikationszyklus kann früher begonnen werden (nämlich nach dem ersten Subframe), was dazu führt, dass die komplette Verarbeitung früher beendet wird. Das wiederum wirkt sich auf den nächsten Kommunikationszyklus aus, der ja genau diesen Zeitpunkt abwarten muss, bevor das Input beginnen kann. Allerdings müssen dafür Subframe IDs dem IFB hinzugefügt werden, was eine Modifikation der PH_{Modes} und der SH_{Modes} erfordert. Zusätzlich muss die CU in geringfügigem Maße geändert werden, so dass der SH_{Mode} aktiviert wird, sobald Daten vom PH_{Mode} an den Data Reader gesendet werden. Eine qualitative Auswertung des Ansatzes ist in Kapitel 3.4 gegeben.

3.3.2 subframebased schedule

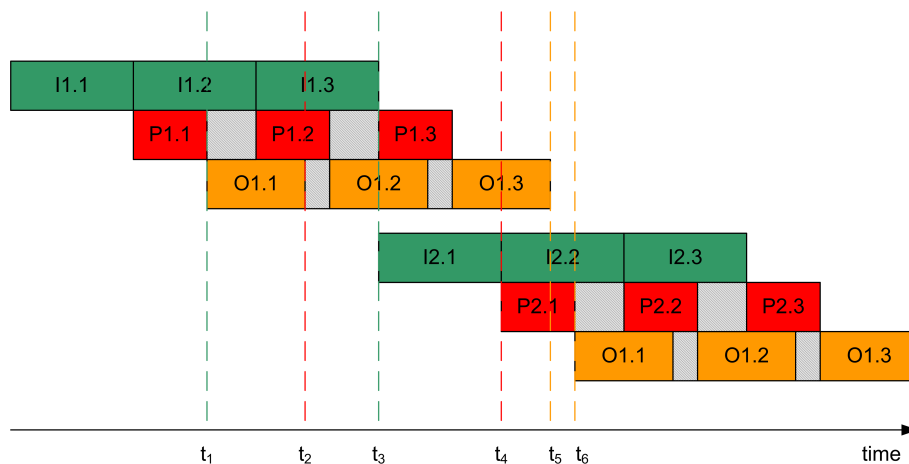


Abbildung 3.7: subframebased Schedule

Im Gegensatz zum framebased Schedule, der Subframe IDs verwendet, wird in diesem Ansatz die eigentliche Struktur des Frames geändert. Die Subframes werden nun zu "unabhängigen" Frames erweitert, indem jeder Subframe um eine **Establish** und eine **Release** Phase erweitert wird. Das führt dazu, dass jeder Subframe des eingehenden Protokolls nun eine Input Stage darstellt (siehe Abbildung 3.8).

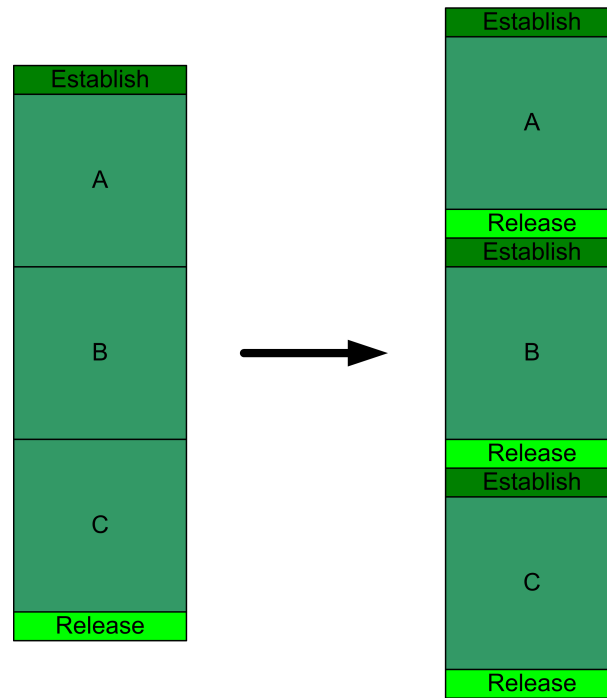


Abbildung 3.8: Links der ursprüngliche Frame, rechts die Subframes als 'unabhängige' Frames

Dies hat den Effekt, dass sich der PH_{Mode} für jeden Subframe um die Belegung des Datenbusses bewirbt und nach jedem Subframe den Datenbus wieder freigibt. Dann bewirbt er sich erneut um den Bus. Diese Vorgehensweise ermöglicht es anderen, höher priorisierten Prozessen, den Datenbus zwischenzeitlich zu belegen, was in zeitbehafteten Protokollen zu Verletzung von Zeitschranken führen kann.

Für dieses Verfahren muss der IFB jedoch nicht modifiziert werden. Die nun unabhängigen Subframes werden als normale Frames eingelesen und behandelt. Es werden auch keine Subframe IDs mehr verwendet, sondern jeder Subframe ist jetzt durch eine normale Frame ID gekennzeichnet.

Input Bedingungen

Die Bedingungen, die in diesem Scheduling gelten, sind vergleichbar mit denen im simple Schedule (da ja nichts anderes als das Einlesen von Frames gemacht wird). Die erste Input Stage eines Kommunikationszyklus (bspw. I2.1) darf dann begonnen werden, wenn sowohl die letzte Input Stage des vorigen Kommunikationszyklus beendet ist (I1.3) als auch das Processing des vorigen Kommunikationszyklus mit der gleichen ID (hier P1.1). Im Beispiel in Abbildung 3.7 ist letztere Bedingung bereits zum Zeitpunkt t_1 erfüllt, aber die erste Input Stage muss noch bis zum Zeitpunkt t_3 warten, da erst die Input Stage aus dem ersten Kommunikationszyklus beendet sein muss. Alle weiteren Subframes (hier I2.2 und I2.3) müssen ebenfalls auf die Beendigung der Processing Stages aus der vorigen Stufe (also P2.2 bzw. P1.3) warten. Zusätzlich muss der vorige Subframe eingelesen und

der Datenbus zum Data Reader wieder freigegeben sein.

- $\text{Input}(x.1)$ nach $(\text{Input}(x-1.L) \text{ und } \text{Process}(x-1.1))$
- $\text{Input}(x.y)$ nach $(\text{Input}(x.y-1) \text{ und } \text{Process}(x-1.y))$ für $y > 1$

Processing Bedingungen

Für die Processing Stages gelten die folgenden Bedingungen: zum Einen müssen die Daten, die in dieser Stage verarbeitet werden sollen, vorhanden sein. Diese Bedingung ist zum Zeitpunkt t_4 erfüllt. Zum Anderen muss die Output Stage aus dem vorigen Zyklus mit der gleichen Subframe Nummer beendet sein (bspw. muss P2.1 auf O1.1 warten). Dies ist in Abbildung 3.7 zum Zeitpunkt t_2 erfüllt. Beide Bedingungen sind durch die roten Striche kenntlich gemacht.

- $\text{Process}(x.y)$ nach $(\text{Input}(x.y) \text{ und } \text{Output}(x-1.y))$

Output Bedingungen

Auch in diesem Verfahren gelten unterschiedliche Bedingungen für den ersten und für die restlichen Subframes. Der Subframe darf erst dann ausgeführt werden, wenn die Daten, die in dieser Output Stage versendet werden sollen, alle vorliegen (O2.1 muss also auf P2.1 warten) und wenn die letzte Output Stage aus dem vorigen Kommunikationszyklus beendet ist (was zum Zeitpunkt t_5 erfüllt ist). Alle anderen Output Stages müssen auf die zugehörige Processing Stage warten und auf das Ende der vorigen Output Stage (bspw. muss O2.2 auf P2.2 und O2.1 warten).

- $\text{Output}(x.1)$ nach $(\text{Process}(x.1) \text{ und } \text{Output}(x-1.L))$
- $\text{Output}(x.y)$ nach $(\text{Process}(x.y) \text{ und } \text{Output}(x.y-1))$ für $y > 1$

Auch durch das subframebased Schedule kann eine Verbesserung der Latenz erzielt werden. Durch das Umwandeln von Subframes in Frames konnte erreicht werden, dass die Verarbeitung noch früher begonnen werden konnte. Es musste nicht das Ende der letzten Processing Stage (bspw. P1.3) abgewartet werden, sondern das Ende von P1.1 und die Freigabe des Datenbusses (I1.3).

In diesem Verfahren ist eine Veränderung des IFBs, wie sie im framebased Schedule notwendig ist, nicht notwendig. Die Verbesserung wird durch das Hinzufügen zusätzlicher **Establish** und **Release** Phasen erreicht.

Im nächsten Abschnitt werden einige konkrete Beispiele präsentiert, um die Verbesserungen, die durch die einzelnen Schedules gewonnen werden, qualitativ abzuschätzen.

3.4 Ergebnisse

Die Ergebnisse, die in diesem Abschnitt präsentiert werden, wurden durch eine Erweiterung des IFS-Editors generiert, die im Rahmen dieser Studienarbeit entwickelt wurde.

Dabei handelt es sich um eine Visualisierung von Scheduling Diagrammen, wie sie in Abbildung 3.9 dargestellt ist.

Um die Beispiele zur Auswertung der beiden vorgestellten Optimierungsverfahren zu generieren, müssen zwei Annahmen bezüglich der Daten im Superframe gemacht werden. Die erste Annahme legt fest, dass die Menge der eingelesenen und versendeten Daten gleich ist. Daraus folgt, dass die Anzahl der Output Instanzen, die ursprünglich den Output-Supeframe gebildet haben, konstant ist. Die jeweilige Verteilung der Bits und damit der Anordnung der Unterteilungspunkte resultiert ausschließlich in einer bestimmten Gruppierung der Output Instanzen. Die zweite Annahme bezieht sich auf eben diese Gruppierungen der Output Instanzen. Dabei wird in den Beispielen davon ausgegangen, dass sich die Instanzen in zwei, vier und acht Gruppen äquivalenter Größe unterteilen lassen.

Die Beispiele sind folgendermaßen aufgebaut:

- es werden zwei Kommunikationszyklen betrachtet
- pro Kommunikationszyklus werden acht Bits vom IFB eingelesen und acht Bits gesendet
- im framebased Schedule und im subframebased Schedule werden diese acht Bits nacheinander in Subframes zu vier, zwei und einem Bit aufgeteilt

Abbildung 3.9 zeigt einen Ausschnitt des simple Schedules, das mit der implementierten Erweiterung des IFS-Editors visualisiert wird.

Eine Betrachtung der zyklengenauen Ausführung (Tabelle 2.1) erlaubt vorweg eine einfache Optimierung. Wie in Abbildung 3.9 sichtbar, konnte noch eine geringfügige Verbesserung implementiert werden: die ersten beiden Schritte der **Establish** Phase im zweiten Zyklus (hier dunkelgrün dargestellt) können bereits *während* des **Processings** (rot) ausgeführt werden (obwohl die Input Bedingung in Kapitel 3.3.1 besagt, dass das **Establish** erst nach dem **Processing** ausgeführt werden dürfte). Dies ist dadurch möglich, dass die beiden ersten Schritte nur die Frame ID und den IRQ setzen (siehe Tabelle 2.1 in Kapitel 2.6). Diese Schritte beeinflussen nicht die Verarbeitung oder den Inhalt des Speichers. Der letzte Schritt, die Anforderung des Datenbusses kann erst dann ausgeführt werden, wenn das **Processing** komplett beendet ist. Mit dieser zusätzlichen Verbesserung kann der zweite Kommunikationszyklus bereits im 56. Takt begonnen werden. Nach 166 Takten sind beide Zyklen beendet. Diese 166 Takte sind der Referenzwert für die beiden Scheduling.

In Abbildung 3.10 wird das **framebased Schedule** dargestellt. Die Input-Frames wurden in zwei Subframes mit je vier Bits aufgeteilt. Durch die Aufteilung kann der erste Zyklus bereits nach 82 Takten beendet werden, der zweite Kommunikationszyklus ist in Takt 139 abgearbeitet. Es liegt also eine Verbesserung von 17% gegenüber dem simple Schedule vor.

Im **subframebased Schedule** mit zwei Subframes a vier Bits (Abbildung 3.11) kann der erste Kommunikationszyklus nach 91, der zweite nach 151 Takten beendet werden. Der subframebased Schedule ist in diesem Fall immer noch besser als der simple Schedule (um 10%), aber er benötigt 12 Takte mehr als der framebased Schedule.

Wird die Größe der Subframes und der Instanzen auf jeweils ein Bit reduziert, ergeben sich noch deutlichere Unterschiede. Das framebased Schedule benötigt für das Einlesen der acht Subframes und dem Senden der acht Instanzen 118 Takte. Gegenüber dem simple Schedule ist das eine Verbesserung von 28%. Der subframebased Schedule hingegen benötigt für die gleiche Anzahl Subframes und Instanzen 205 Takte. Er ist also um 39 Takte (das entspricht 24%) langsamer als der simple Schedule.

Folgende Tabelle stellt diese und weitere Werte gegenüber.

Verfahren	8 Bits	2x4 Bits	4x2 Bits	8x1 Bit
simple Schedule	166	—	—	—
framebased Schedule	166	139	124	118
subframebased Schedule	166	151	199	205

Tabelle 3.1: Vergleich der Ausführungszeiten der Optimierungsverfahren in IFB-Takten für einen Frame mit 8 Bit

Das framebased Schedule erzielt also bessere Ergebnisse als das simple Schedule. Je kleiner die Subframes und die Instanzen, desto schneller können sie abgearbeitet werden. Diese Verbesserungen erfordern aber eine Modifikation des IFB (siehe Kapitel 3.3.1).

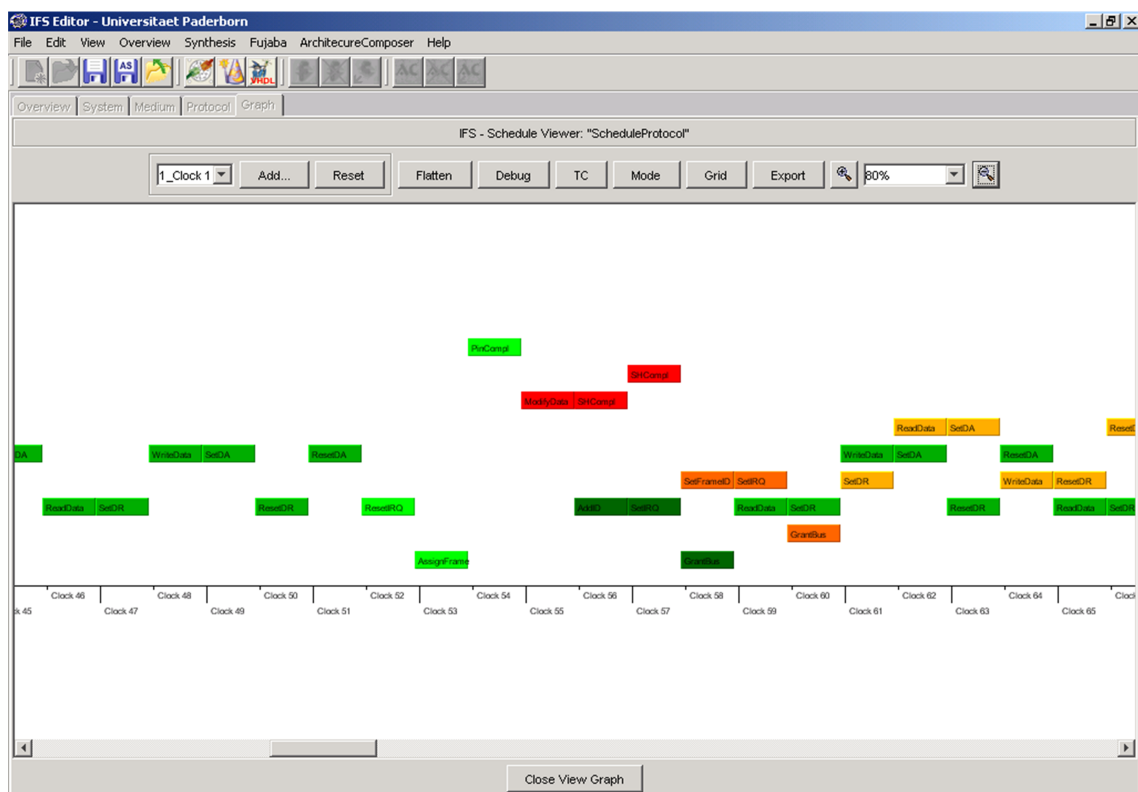


Abbildung 3.9: simple Schedule mit 8 Bits pro Frame

3 Datenflussoptimierung

Das subframebased Schedule erzielt nur bis zu einer Subframegröße von vier Bits ein besseres Ergebnis als der simple Schedule. Sind die Subframes kleiner, benötigt das Durchlaufen der zusätzlichen **Establish** und **Release** Phasen benötigt mehr Zeit, als durch die Verbesserung gewonnen werden kann. Eine Modifikation des IFB ist in diesem Scheduling nicht erforderlich, allerdings sind die Verbesserungen geringer als im framebased Schedule. Betrachtet man große Input-Frames, deren Daten auf wenige Gruppen abgebildet werden, so ist das Ergebnis nur geringfügig schlechter als das des framebased Schedules. Ein Beispiel mit 1024 Input Bits ist in Tabelle 3.2 dargestellt und belegt diesen Effekt.

Verfahren	1024 Bits	2x512 Bits	4x256 Bits	8x128 Bit
simple Schedule	18.455	–	–	–
framebased Schedule	18.455	15.377	13.840	13.060
subframebased Schedule	18.455	15.391	13.878	13.150

Tabelle 3.2: Vergleich der Ausführungszeiten der Optimierungsverfahren in IFB-Takten für einen Frame mit 1024 Bit

Wie gezeigt wurde, kann durch beide Verfahren eine deutliche Verbesserung der Latenz erreicht werden. Die vorgezogene Verarbeitung der Daten ist somit ein effektiver

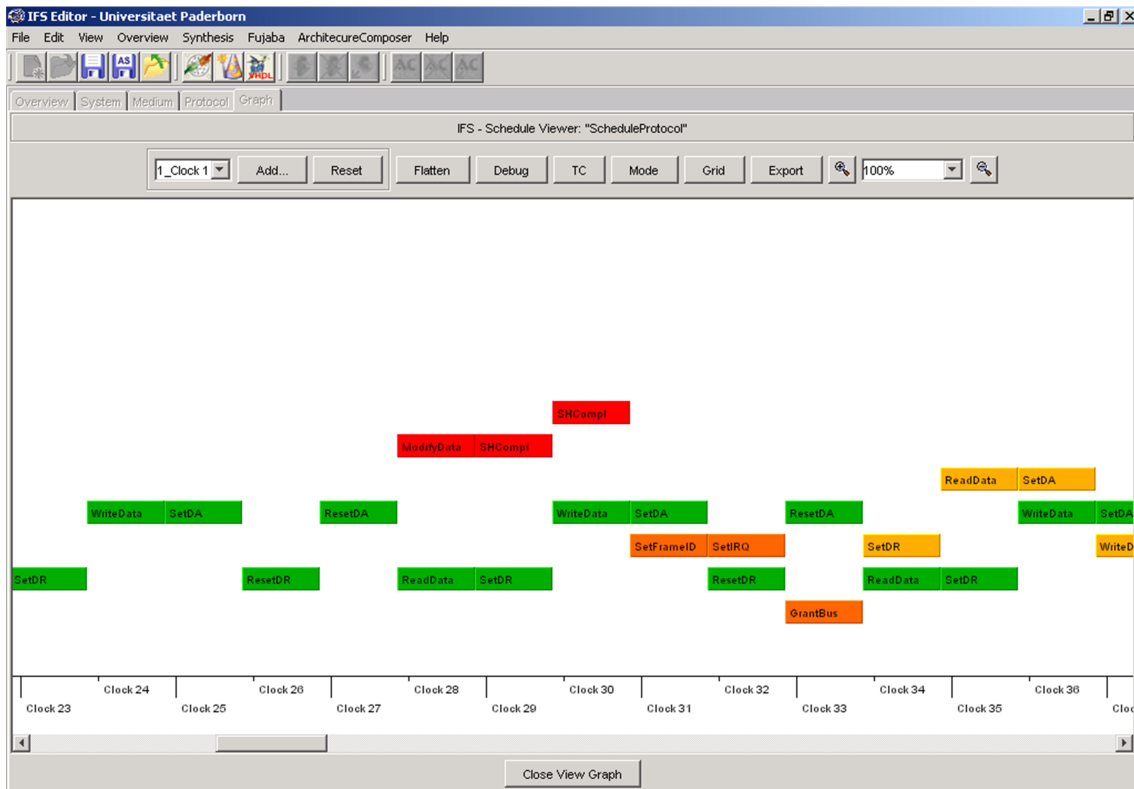


Abbildung 3.10: framebased Schedule mit 2 Subframes mit vier Bits

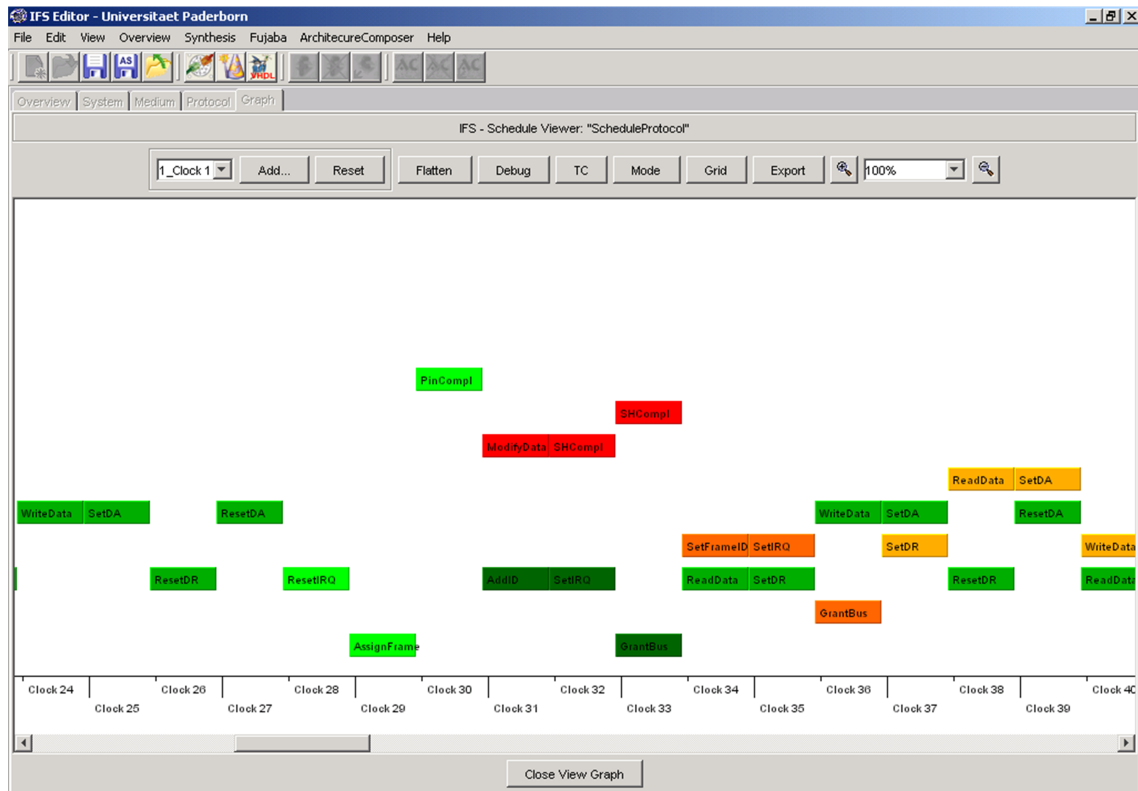


Abbildung 3.11: subframebased Schedule mit 2 Subframes mit vier Bits

Optimierungsansatz. Die Ergebnisse beider Verfahren hängen dabei von dem Anwendungszenario ab.

4 Rekonfigurierte Ausführung

Im vorigen Kapitel wurde gezeigt, wie durch geschicktes Pipelining im IFB die Latenz des IFBs reduziert werden konnte. Dies geschah unter der Annahme, dass alle Modes, die benötigt werden, immer auf dem Chip vorhanden sind. Geht man nun davon aus, dass ein IFB mit vielen Tasks verbunden ist und viele Mapping Equations implementiert, so werden entsprechend viele Modes benötigt. Somit benötigt die Implementierung des IFB viel Platz. Der Platz stellt auf einem Chip aber eine teure Ressource dar und Chips können nicht beliebig gross konstruiert werden. Die damit verbundenen Kosten und die Größe der Chips würden enorm steigen, was in dem meisten Fällen unerwünscht ist. Man möchte eher "kleine" Chips verwenden.

Hier setzt der zweite Optimierungsansatz an. Es wird davon ausgegangen, dass nicht ausreichend Platz für alle Modes vorhanden ist, die von einem IFB benötigt werden. Somit ergibt sich die Notwendigkeit, Modes zu verdrängen und andere Modes zu laden. Das wiederum erfordert Hardware, die zur Laufzeit rekonfiguriert werden kann. Es werden also multifunktionale Ausführungseinheiten benötigt, in die dynamisch die benötigten Komponenten geladen und dann ausgeführt werden können. Dazu wird ein FPGA verwendet.

Wie bereits im Abschnitt 2.1 kurz erläutert, besteht das FPGA aus *slices*, die wiederum gruppiert und zu Slots zusammengefasst werden können (siehe Abbildung 2.1). Diese Slots sind dynamisch rekonfigurierbar, d.h., dass zur Laufzeit vorhandene Modes dadurch verdrängt werden, indem neue Modes in diese Slots geladen werden. Hierbei unterscheidet man zwischen *multi reconfigurable* (es können mehrere Modes gleichzeitig geladen werden) und *single reconfigurable* (es kann nur ein Mode gleichzeitig geladen werden) FPGAs.

Zur Rekonfigurierung wird eine Kontrollkomponente benötigt, die den Vorgang der Rekonfigurierung ausführt und kontrolliert. Denn bevor eine Rekonfigurierung ausgeführt werden kann, muss die Einhaltung bestimmter Bedingungen sichergestellt werden. Es ergeben sich folgende Aufgaben für die Kontrolleinheit:

- Auswahl des Modes, der verdrängt wird
- Sicherstellung, dass zu Beginn der Rekonfigurierung kein Mode aktiv ist und somit keine Kommunikation oder Verarbeitung im Gange ist
- Das korrekte Einbinden des neuen Modes in den Slot (u.a. die Verbindungen)
- Das Anpassen des Interfaces an die neuen Funktionalitäten (wenn nötig)
- Das Sicherstellen eines reibungslosen Ablaufes ohne Beeinflussung der anderen Modes (ohne dass bei diesen Datenverlust auftritt)

Diese Kontrollkomponente ist für das FPGA notwendig und darf nicht ausgetauscht werden. Deswegen ist sie auf dem *fixed slot* implementiert. Dieser Slot ist nicht rekonfigurierbar, d.h. diese Slots können nicht durch andere Funktionalitäten überschrieben werden.

Im Falle des IFBs werden auf dem nicht rekonfigurierbaren Slot neben der Kontrolleinheit (der *Reconfiguration Unit*) weitere Komponenten untergebracht, die für die Funktionalität des IFBs unabdingbar sind und somit nicht ausgetauscht werden dürfen:

Die **Control Unit** mit ihren Unterkomponenten (dazu gehört die *Reconfiguration Einheit*) sind wichtige Steuerinstrumente zur Kontrolle des IFBs.

Der **Speicher** (Data Reader und Data Writer) dient zur Zwischenablage der Daten aus den und für die Modes. Alle Komponenten müssen darauf zugreifen können. Ein Austausch des Speichers würde somit zu Datenverlust führen.

Auf dem festen Slot sind außerdem die **Verbindungsleitungen** zu den externen Tasks untergebracht. Letztere sind nicht direkt mit den Slots verbunden, in denen die PH_{Modes} geladen sind, sondern die Verbindung läuft erst über diesen nicht austauschbaren Slot. Von hier aus gehen die Leitungen zum PH_{Mode} . Dies hat den Vorteil, dass das Interface für die externen Tasks immer das gleiche ist und ein Mode in einen beliebigen Slot geladen werden kann. Somit ist eine Anpassung der Tasks bzw. des FPGAs unnötig (und auch nicht wünschenswert), denn der IFB soll für die Tasks ja transparent bleiben.

Somit ergibt sich die bereits bekannte Abbildung 2.1.

Um die Rekonfigurierung von Modes zu beschreiben, wurde das bisher verwendete Modell erweitert. Grundlage war das I-P-O Schema. Es umfasste nur das Ausführen der Input Stage, der Processing Stage und der Output Stage. Ist eine dieser Stages aber nicht vorhanden, muss sie erst in einen Slot geladen werden. Also muss für jede der drei Stufen noch eine weitere Stufe hinzugefügt werden, in der die entsprechende Komponente in einen Slot geladen wird. Es ergibt sich das folgende Schema:

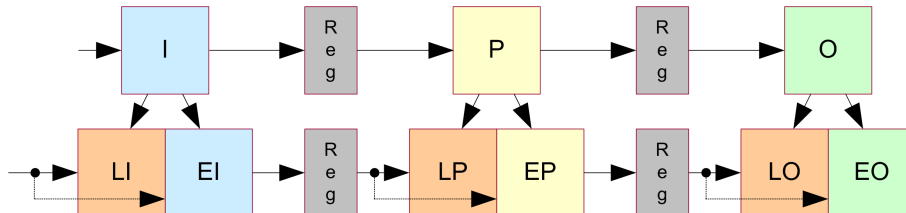


Abbildung 4.1: Das neue I-P-O Schema mit sechs Stufen

Abbildung 4.1 zeigt das daraus resultierende Schema für die Ausführungs-Pipeline. Die bereits bekannten Stufen wurden in eine *Load* Phase und eine *Execute* Phase aufgeteilt. Zu beachten ist, dass diese neuen Stufen durchaus übersprungen werden können (durch den Pfeil gekennzeichnet, der durch die *Load* Phasen geht). Ist eine Komponente bereits geladen, ist es unnötig, diese nochmals zu laden.

Wie bereits in der Einleitung erwähnt, ist dieser Ansatz nicht mit der Datenflussoptimierung kombinierbar. Dieser betrachtet alle benötigten Modes als gegeben an, d.h. sie sind von alle auf dem FPGA geladen und können ohne Verzögerung ausgeführt werden.

Das Laden einer Stage wurde somit nicht berücksichtigt. Diese Optimierung kann also genutzt werden, wenn die Chips, auf denen ein IFB implementiert wird, genügend groß sind, so dass tatsächlich alle Modes bereits zu Beginn geladen werden können. Benötigt der IFB aber viele Modes (da viele Tasks an ihn angeschlossen sind), für die nicht ausreichend Slots zur Verfügung stehen, kann die Flächenoptimierung genutzt werden. Diese Optimierung sorgt für eine optimale Ausnutzung (Utilization) der Slots, so dass Stages frühestmöglich und in optimaler Reihenfolge geladen werden.

Nachdem das erweiterte Modell eingeführt und erklärt wurde, wird gezeigt, wann welche Stage ausgeführt werden kann und welche Bedingungen eingehalten werden müssen, um einen korrekten Ablauf sicherzustellen.

4.1 I-P-O Scheduling eines rekonfigurierbaren IFB

Die grundlegende Idee beim Scheduling des erweiterten I-P-O Schemas ist das Caching. Hat bspw. ein PH_{Mode} die Daten einer Task gelesen, so wird dieser Mode nicht sofort verdrängt. Er bleibt vorerst geladen, denn dieser Mode wird im nächsten Kommunikationszyklus wieder benötigt. Der Mode bleibt also so lange in einem Slot geladen, bis er von einem anderen verdrängt werden muss. Ein Beispiel soll dies verdeutlichen:

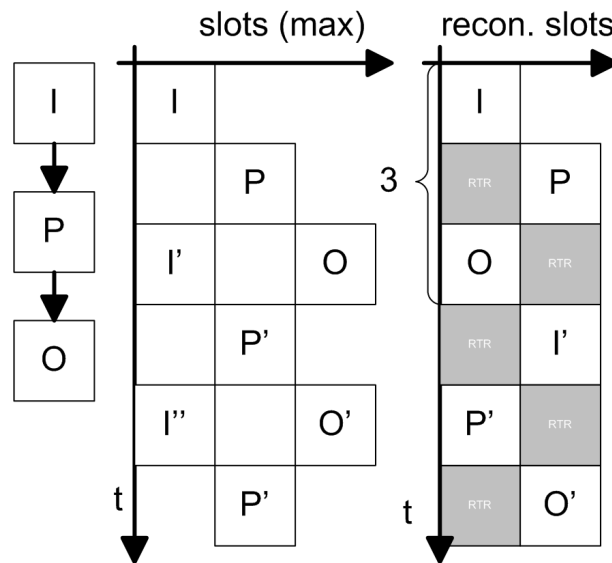


Abbildung 4.2: Scheduling mit beliebig vielen Slots (links) und zwei Slots (rechts) [Ihm05].

Im linken Teil von Abbildung 4.2 stehen die maximal benötigte Anzahl an Slots (in diesem Fall drei) zur Verfügung, so dass die Ausführung des IFBs ohne unnötige Verzögerungen stattfindet. Das ist dadurch möglich, dass alle Slots parallel geladen werden und ein Nachladen zur Laufzeit nicht notwendig ist.

Im rechten Teil wird deutlich, dass weniger Slots die Ausführung des IFBs verzögern, da Stages zur Laufzeit nachgeladen werden müssen (RTR = *Runtime Reconfiguration*). Dies führt zu einem Trade-Off zwischen Rechenzeit und benötigtem Platz. Im dritten Takt kann daher nur die Output Stage des ersten Kommunikationszyklus ausgeführt werden. Im anderen Beispiel war es möglich, parallel zur Output Stage die Input Stage für den nächsten Kommunikationszyklus im zweiten Takt zu laden und im dritten Takt auszuführen. Um eine Verdrängung durchzuführen, wird eine Verdrängungsstrategie benötigt, die im Folgenden erklärt wird [Hen02]:

1. grundsätzlich werden nur Modes verdrängt, die nicht aktiv sind
2. existieren keine inaktive Modes, wird die Verdrängung verzögert
3. existiert mindestens ein inaktiver Mode, so werden folgende Fälle unterschieden:
 - a) ist genau ein Mode inaktiv, wird dieser ersetzt
 - b) sind mehrere Modes inaktiv, wird der ersetzt, der am weitesten entfernt und von dem verdrängenden Mode in seiner Ausführung abhängig ist
 - c) existiert kein abhängiger Mode, so wird einer der Modes mit der größten Entfernung verdrängt

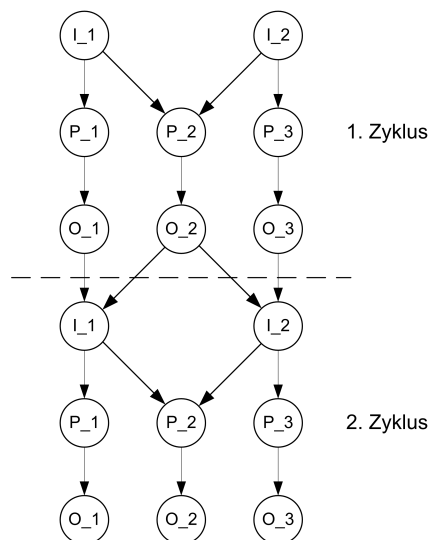


Abbildung 4.3: Die Stages zweier Kommunikationszyklen sowie deren Abhängigkeiten

Diese Verdrängungsstrategie ist optimal. Die Grundlage des Verfahrens ist aus der Caching Theorie abgeleitet, die besagt, dass idealerweise das am längsten nicht mehr benötigte Objekt zu verdrängen ist. Der Graph in Abbildung 4.3 zeigt beispielhaft zwei Kommunikationszyklen. Die Knoten des Graphen repräsentieren die Stages, die ausgeführt werden müssen. Der Abstand zwischen zwei Stages lässt sich in Form der vertikalen Distanz zwischen der Stage, die platziert werden soll, und der Stage, die in einem Slot allokiert ist, messen. Das folgende Kapitel enthält einige Ergebnisse zu dieser Strategie.

4.2 Ergebnisse

Um die Effektivität der oben präsentierten Verdrängungsstrategie aufzuzeigen, wurden verschiedene Beispiele im IFS-Editor simuliert.

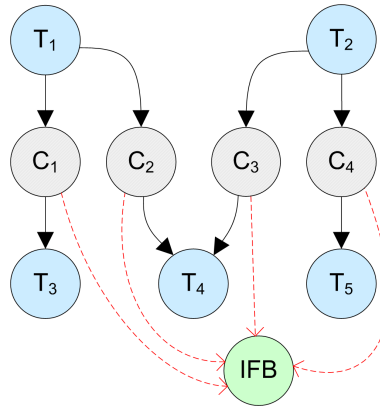


Abbildung 4.4: Der Kommunikationsgraph aus der Einleitung

In Abbildung 4.4 wurde der bereits bekannte Kommunikationsgraph nochmals gezeigt. In diesem Beispiel existieren zwei sendende und drei empfangende Tasks. Zwischen den Tasks sind die Abbildungsregeln, die auf den IFB abgebildet werden. Die folgenden Abbildungen zeigen beispielhaft das I-P-O Scheduling des in Abbildung 4.4 abgebildeten Szenarios. Dabei wurde ein FPGA mit vier Slots simuliert und der Kommunikationszyklus zweimal ausgeführt.

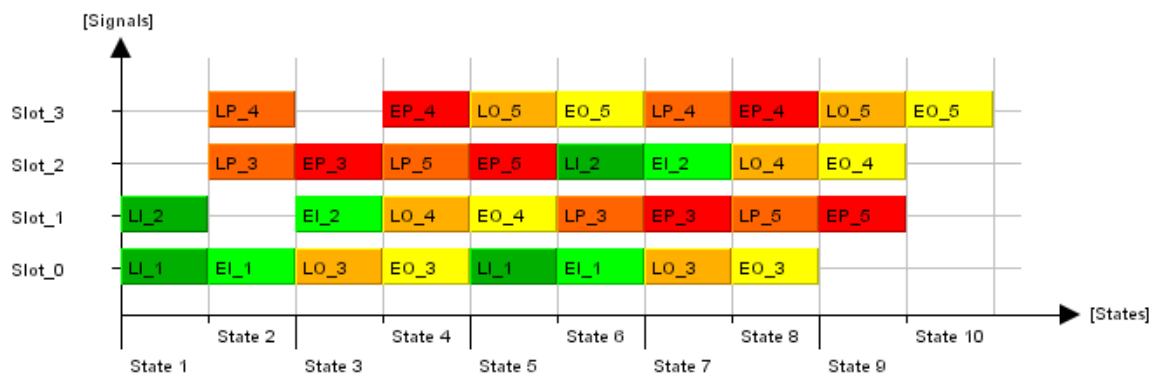


Abbildung 4.5: Zwei Kommunikationszyklen auf einem multi reconfigurable FPGA

4 Rekonfigurierte Ausführung

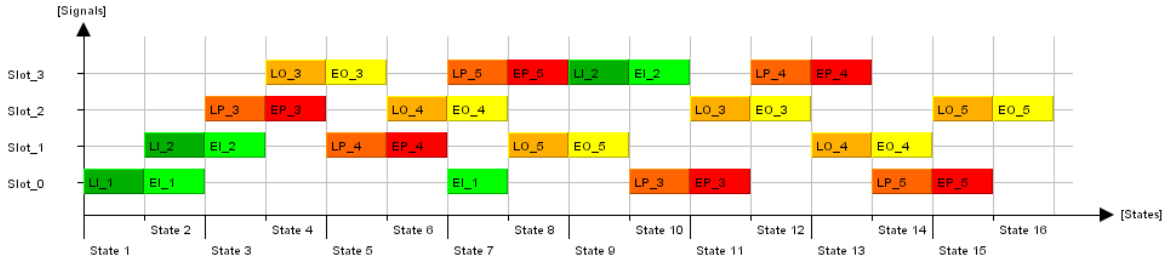


Abbildung 4.6: Zwei Kommunikationszyklen auf einem single reconfigurable FPGA

Abbildung 4.5 zeigt dabei die Ausführung auf einem multi reconfigurable FPGA. In Abbildung 4.6 wurde ein single reconfigurable FPGA verwendet. Für das Beispiel wurden vier Slots verwendet, weil hier die Unterschiede zwischen den beiden Varianten besonders deutlich hervortreten. So benötigt das multi reconfigurable FPGA sechs Takte weniger und weist dabei eine höhere Utilization (siehe Gleichung 4.1) auf.

Im Folgenden werden drei repräsentative Szenarien betrachtet, die eine qualitative Bewertung des vorgestellten Pipelining-Verfahrens erlaubt:

- 1x Input, 3x Output (ein sendender und drei empfangende Tasks)
- 2x Input, 3x Output (siehe Abbildung 4.4)
- 3x Input, 1x Output

Die folgenden drei Abbildungen zeigen die Ergebnisse der berechneten Scheduling der drei Szenarien. Dabei werden sowohl die Anzahl der benötigten Schritte (*Pipeline Clocks*) als auch die Flächenausnutzung auf dem Chip (*Utilization*) für multi reconfigurable und single reconfigurable FPGAs visualisiert. Die Utilitization berechnet sich mit der Formel

$$Utilization = \frac{\sum_{states} \text{Anzahl aktiver Slots}}{\text{Anzahl aller Slots} \cdot \text{Anzahl der Slots}} \quad (4.1)$$

Jedes Szenario wurde erst mit einem Slot, dann mit zwei Slots, usw. ausgeführt. Sobald in einem Scheduling die Anzahl der Slots größer oder gleich der Anzahl der zu ladenden Stages ist, kann keine Verbesserung der Pipeline clocks erzielt werden. Daher war sind die dargestellten Diagramme auf zehn Slots begrenzt. Zusätzliche Slots würden nur zu einer Verschlechterung der Utilization führen.

Jedes dieser Szenarien wird zehn Mal hintereinander ausgeführt, es ergeben sich also zehn Kommunikationszyklen.

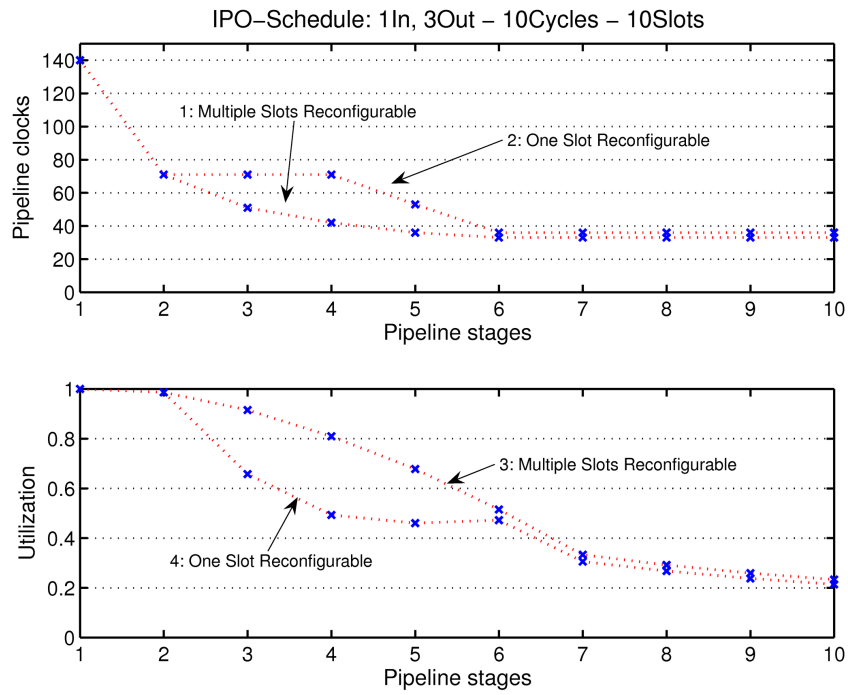


Abbildung 4.7: IPO-Schedule: Ein sendender und drei empfangende Tasks

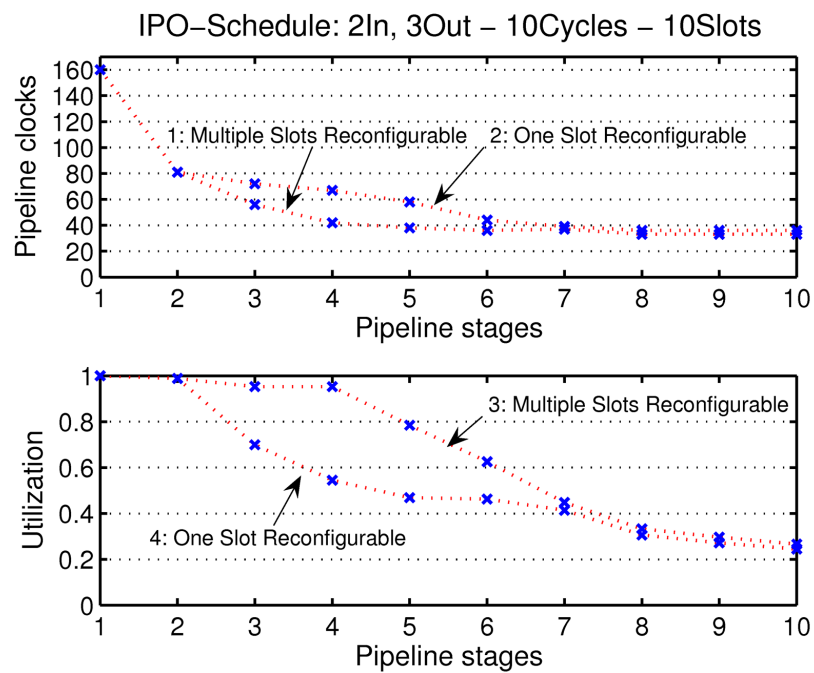


Abbildung 4.8: IPO-Schedule: Zwei sendende und drei empfangende Tasks

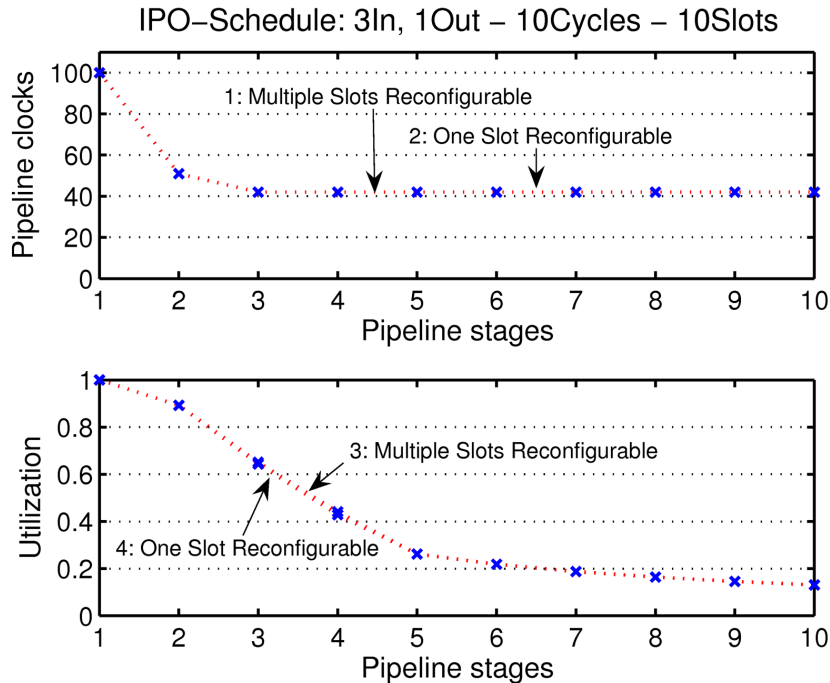


Abbildung 4.9: IPO-Schedule: Drei sendende und ein empfangender Tasks

In den ersten beiden Szenarien (Abbildungen 4.7 und 4.8) ist deutlich zu erkennen, dass es von Vorteil ist, wenn mehrere Slots gleichzeitig rekonfiguriert werden können. Somit wird sowohl die Zahl der benötigten Schritte reduziert als auch der Platz optimaler ausgenutzt (es werden sind also mehr Stages pro Pipeline Clock geladen). Diese Unterschiede treten hauptsächlich auf, wenn zwei bis sechs Slots zur Verfügung standen, um die Pipeline Stages zu laden. Mit mehr oder weniger Slots weichen die erzielten Ergebnisse nicht gravierend voneinander ab. Es ist also von Vorteil, mit zwei bis sechs verfügbaren Slots einen multi reconfigurable FPGA zu verwenden.

In Abbildung 4.9 zeigen sich keine deutlichen Unterschiede zwischen beiden Versionen. In diesem speziellen, sehr einfachen Beispiel spielt es keine Rolle, ob mehrere Slots gleichzeitig rekonfiguriert werden können.

Vergleicht man die drei Szenarien hinsichtlich benötigter Zeit und Flächenausnutzung, so wird deutlich, dass je mehr Slots zur Verfügung stehen, die **Zeiten** verbessert werden können. Allerdings konvergieren die Zeiten für die ersten beiden Szenarien stetig gegen die minimale Anzahl von Pipeline Clocks, sobald mehr als sechs Slots verfügbar sind. Danach können keine weiteren Verbesserungen erzielt werden. Beim dritten Szenario (Abbildung 4.9) pendelt sich dieser Wert bei etwas mehr als 40 Zeiteinheiten ein. Die **Flächenausnutzung** dagegen nimmt unterschiedlich stark ab, erreicht aber bei allen drei Szenarien mit zehn Slots einen Wert zwischen 15% und 25%.

Zusammenfassend kann gesagt werden, dass multi reconfigurable FPGAs mit zwei bis vier Slots deutliche Vorteile besitzen. Damit können die Kommunikationszeiten reduziert

und die Fläche besser ausgenutzt werden, ohne die Kommunikationszeiten drastisch zu erhöhen. Voraussetzung für eine solche Optimierung ist, dass keine Real-Time Protokolle ausgeführt werden können, da sonst ggf. Deadlines verletzt werden könnten.

Diese Optimierung ermöglicht es, die benötigte Fläche für eine IFB-Implementierung zu reduzieren. Das erlaubt die Verwendung kleinerer Chips, was wiederum die Kosten des Designs senkt. Die Rekonfigurierungseigenschaften des Chips beeinflussen dabei die erreichbaren Verbesserungen. Plattformen, auf denen mehrere Teilschaltungen gleichzeitig rekonfiguriert werden können, bieten Vorteile gegenüber solchen Plattformen, die nur die Rekonfigurierung eines Teilstücks unterstützen.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

In dieser Arbeit wurden zwei Verfahren präsentiert, um die Ausführung des IFBs zu beschleunigen bzw. zu optimieren.

Das erste Verfahren zielte auf die Reduzierung der Latenz durch Analyse der eingelesenen Daten. Die Verarbeitung wurde nicht erst nach dem Einlesen *aller* Daten gestartet, sondern ein Teil der Daten konnte bereits vorher modifiziert werden. Dazu wurden die Subframes eingeführt, die auf zwei unterschiedlichen Arten in den IFB integriert wurden. Das erste Verfahren (das framebased Schedule) erwies sich als sehr effizient. Je kleiner die Subframes und die Instanzen wurden, desto schneller konnten die einzelnen Stages durchlaufen und beendet werden. Das zweite Verfahren (subframebased Schedule) stellte erwies sich ebenfalls als effizient. Nur für Subframegrößen unter vier Bit wurden schlechtere Ergebnisse erzielt, was keine wesentliche Einschränkung darstellt.

Das zweite Optimierungsverfahren beschreibt die Implementierung einer effektiven Verdrängungsstrategie, um die benötigte Fläche für die Implementierung eines IFB zu minimieren. Dazu werden die einzelnen Stages des I-P-O Schemas in **Load Stage** und **Execute Stage** aufgeteilt und eine Verdrängungsstrategie verwendet, die diese Aufteilung berücksichtigt. In Kapitel 4.2 wurde gezeigt, dass ein FPGA die besseren Ergebnisse erzielt, wenn mehrere Slots gleichzeitig rekonfiguriert werden können. Außerdem ist diese Strategie dann am effektivsten, wenn weniger Sender als Empfänger vorhanden sind.

Beide Strategien sind aus den bereits dargelegten Gründen nicht miteinander kombinierbar. Das bedeutet, dass man eine Entscheidung hinsichtlich der Bedeutung von Latenz und Fläche treffen muss. Ist eine niedrige Latenz von hoher Bedeutung, so sollte das framebased Schedule implementiert werden. Verfügt man aber nur über wenige Slots (ist also der FPGA relativ klein und kann nicht alle Modes gleichzeitig speichern), so sollte man die Flächenoptimierung mit der Verdrängungsstrategie aus Kapitel 4.1 verwenden.

5.2 Ausblick

In dieser Arbeit wurde für die Datenflussoptimierung nur der einfache Fall mit einer Input Stage, einer Processing Stage und einer Output Stage berücksichtigt. Ein mögliches Szenario besteht darin, dass eine Processing Stage Daten von *zwei* Input Stages benötigt. Das Scheduling für einen solchen Fall könnte durch eine Erweiterung der Scheduling Algorithmen implementiert werden.

Literaturverzeichnis

- [Ave05] D. Averberg. Synthese von deadline-konformen protokollkonvertern für heterogene verteilte anwendungen. Studienarbeit, Universität Paderborn, März 2005.
- [Hen02] Hennessy, John L. and Patterson, David A. and Goldberg, David und Asanovic, Krste. *Computer Architecture*. Morgan Kaufmann, June 2002.
- [Ihm02] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. A Consistent Design Methodology for Configurable HW/SW-Interfaces in Embedded Systems. Montreal, Canada, Aug. 2002.
- [Ihm03] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. Modeling of Configurable HW/SW-Interfaces. In *RSS2003*, pages 51 – 60, 2003.
- [Ihm04] Ihmor, Stefan and Hardt, Wolfram. Runtime Reconfigurable Interfaces - The RTR-IFB Approach. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW'04)*. IEEE Computer Society, 26 - 27 April 2004.
- [Ihm05] Ihmor, Stefan and Dittmann, Florian. Optimizing Interface Implementation Costs Using Runtime Reconfigurable Systems . In , 2005.
- [Lok05] T. Loke. Synthese von kommunikationsstrukturen in eingebetteten systemen. Studienarbeit, Universität Paderborn, März 2005.

Synthese von Kommunikationsstrukturen in verteilten Systemen

Tobias Loke

Eingebettete Systeme erobern in der heutigen Zeit immer neue Anwendungsgebiete und übernehmen dabei immer komplexere Aufgaben. Die Kommunikation in verteilten eingebetteten Systemen übernimmt dabei eine zunehmend zentrale Rolle. Die Optimierung der Interaktion heterogener Anwendungen in solchen Systemen steht bei der hier vorgestellten Arbeit im Vordergrund.

Konkret wird ein optimierender Algorithmus zur Platzierung von Interface Blöcken, welche die Protokolladaptierung der interagierenden Komponenten durchführen, auf Bauelemente einer Systemarchitektur beschrieben. Zu diesem Zweck findet ein speziell entwickeltes Clusteringverfahren Anwendung, mit dem Ziel die globalen Kommunikationskosten sowie die benötigten Implementierungsressourcen zu optimieren. Der komplexe Entwurfsraum wird dabei mit Hilfe einer multidimensionalen Kostenfunktion untersucht.

Als weitere Maßnahme zur Reduktion des Kommunikationsoverheads werden "Dedicated Protocols" eingeführt. Dabei handelt es sich um die Verschmelzung mehrerer Protokolle zu einem anwendungsspezifischen Protokoll, welches die Daten der verschmolzenen Protokolle unter Einhaltung aller Bedingungen der einzelnen Protokolle übertragen kann.

Inhaltsverzeichnis

1	Einführung	159
1.1	Motivation	159
1.2	Aufgabenstellung	160
1.3	Aufbau der Arbeit	160
2	Grundlagen	163
2.1	Protokolle	163
2.2	Graphentheorie	163
2.3	Protokoll Grundblöcke	165
2.4	Clustering	166
2.4.1	Hierarchisches Clustering	167
2.4.2	Partitionierendes Clustering	169
3	Interface Synthesis Design Flow	171
3.1	Motivation für automatisiertes Design	171
3.2	Spezifikationsgraph	174
3.3	Kommunikationsgraph	174
3.4	Das IFD-Mapping	174
3.5	Der Aufbau des Interface Blocks (IFB)	176
3.5.1	Control Unit (CU)	176
3.5.2	Protocol Handler (PH)	177
3.5.3	Sequence Handler (SH)	178
4	Automatische Verteilung	179
4.1	Systemtopologien unter Verwendung von IFBs	179
4.1.1	Verwendung eines einzelnen IFBs	179
4.1.2	Verwendung mehrerer IFBs	180
4.2	Automatische Verteilung von IFBs	181
4.2.1	Kostenfunktionen	181
4.2.2	Strategien für die Verteilung von IFBs	183
4.2.3	Clustering als Verfahren zur Reduktion der IFB-Anzahl	184
4.3	Ablauf der Verteilung von IFBs	185
4.3.1	Erstellen einer initialen Verteilung	186
4.3.2	Optimierung	186
4.3.3	Auswahl der Strategie	187
4.3.4	Clustering Algorithmus	187

4.4	Integration im IFS-Editor	188
5	Dedicated Protocol	189
5.1	Ansätze für die Verwendung eines Dedicated Protocols	189
5.2	Merged Protocol als Realisierung eines Dedicated Protocols	189
5.3	Protokoll Parser	191
5.4	Berechnung eines Merged Protocols	193
5.5	Auswirkung auf den IFB	196
6	Zusammenfassung und Ausblick	199
6.1	Zusammenfassung der Arbeit	199
6.2	Ausblick auf weitere Modellierung	199
6.3	Ausblick für mögliche Erweiterungen	200
	Literaturverzeichnis	201

1 Einführung

1.1 Motivation

Der Markt für eingebettete Systeme wächst sehr stark und wird nach Einschätzung von Experten bald den Markt für terminal-basierte Anwendungen übertreffen. Eingebettete Systeme findet man heute in fast allen Bereichen des täglichen Lebens, daher stellen sie einen bedeutenden wirtschaftlichen Faktor dar.

In eingebetteten Systemen steigen die Anforderungen an die Funktionalität immer weiter. Zur Realisierung der benötigten Funktionalität bei einer möglichst kurzen Entwicklungszeit, erlangt das so genannte *IP-based Design (Intellectual Property)* immer stärker an Bedeutung. Bei dem IP-based Design werden bereits vorhandene und getestete Komponenten (*IPs*) wieder verwendet. Daher ist die Integration von bereits vorhandenen Komponenten ein wesentlicher Bestandteil des Entwurfsprozesses von eingebetteten Systemen.

In verteilten eingebetteten Systemen wird so eine komplexe Funktionalität häufig durch die Komposition mehrerer Teilkomponenten realisiert. Dies erfordert eine Interaktion der Teilkomponenten. Die interagierenden Komponenten eines eingebetteten Systems sind die Bestandteile der so genannten *Kommunikationsstruktur*. Bei verteilten Systemen mit einer Vielzahl an Kommunikationskomponenten steigt die Anzahl der möglichen Kommunikationsstrukturen sehr schnell.

Eine weitere Herausforderung im Entwurfsprozess von eingebetteten Systemen stellt die Heterogenität der Komponenten dar, denn es existiert kein einheitlicher Standard unter den Herstellern von IPs. Verfahren zur Unterstützung der Entwickler durch einen möglichst automatisierten Design Prozess gewinnen daher immer mehr an Bedeutung. Das Problem bei der Integration inkompatibler Komponenten wird oft auf eine der beiden folgenden Arten umgangen:

1. Es wird auf die Verwendung inkompatibler Komponenten verzichtet. Dieser Ansatz führt dazu, dass nur Komponenten von einem Hersteller verwendet werden können. Die Auswahl der Komponenten ist dadurch stark eingeschränkt.
2. Zur Interaktion zwischen den Komponenten werden standardisierte Bussysteme verwendet. Jede Komponente, die in eine Kommunikationsstruktur aufgenommen werden soll, muss um eine entsprechende Schnittstelle für ein solches Bussystem erweitert werden. Dies ist ein fehleranfälliger und aufwendiger Bestandteil der Entwicklung. Bei kleineren Komponenten führt er zu einem ungünstigen Verhältnis zwischen der Größe der Schnittstelle für das Bussystem und der eigentlichen IP.

Beide Möglichkeiten sind entweder unflexibel oder bringen einen hohen Aufwand mit sich. Zur Lösung dieses Problems wurde der Ansatz des *Interface Synthesis Design Flows* geschaffen. Er ermöglicht mit Hilfe einer Adapterstruktur, dem so genannten *Interface Block (IFB)*, die Einbindung inkompatibler Komponenten, ohne die Komponenten selbst ändern zu müssen. Bei der Erstellung von komplexeren eingebetteten Systemen, in denen eine Vielzahl von Komponenten an den Kommunikationsstrukturen beteiligt sind, zeigen sich Einschränkungen bei Verwendung des IFS-Flows:

- Die Kommunikationsstruktur kann im IFS-Flow jeweils nur um eine Gruppe von interagierenden Komponenten erweitert werden. Dies ist dadurch bedingt, dass der jeweils benötigte IFB einzeln erstellt werden muss. Es können nicht alle Kommunikationsstrukturen auf einmal erstellt werden.
- Die Erzeugung von Kommunikationsstrukturen mit Hilfe des IFS-Flows erlaubt eine Vielzahl von möglichen Lösungen für die Architektur aus interagierenden Komponenten und IFBs. Die Lösungen unterscheiden sich hinsichtlich ihrer Qualität im Bezug auf den Verbrauch von Ressourcen in der gegebenen Systemarchitektur. Die vorhandenen Ressourcen entscheiden darüber, ob eine komplexe Kommunikationsstruktur überhaupt realisiert werden kann.

Die aufgeführten Einschränkungen des IFS-Flows resultieren in einem reduzierten Automatisierungsgrad.

1.2 Aufgabenstellung

Für die optimierte Synthese von Kommunikationsstrukturen in verteilten eingebetteten Systemen wird ein erweiterter IFS-Flow benötigt. Im Rahmen dieser Studienarbeit sind Verfahren aufzuzeigen, mit denen die beschriebenen Einschränkungen des IFS-Flows kompensiert werden können.

Für die Synthese der vollständigen Kommunikationsstruktur ist ein Verfahren zur automatisierten Verteilung von IFBs zu erstellen. Das Verfahren muss möglichst effizient mit den vorhandenen Ressourcen umgehen, und z.B. die Länge von Verbindungen zwischen Komponenten minimieren.

Neben der automatisierten Verteilung ist ein Ansatz für anwendungsspezifische Protokolle (*Dedicated Protocols*) zu erstellen. Dabei sollen mehrere gegebene Protokolle zu einem Dedicated Protocol verschmolzen werden, um den Ressourcenverbrauch auf Kommunikationswegen zu verringern und somit komplexere Kommunikationsstrukturen zu ermöglichen.

1.3 Aufbau der Arbeit

Nach dieser Einleitung folgt das Kapitel zwei, in dem die verwendete Definition von Protokollen angegeben ist. Anschließend werden Begriffe aus der Graphentheorie vorgestellt,

die später im Kapitel fünf verwendet werden. Am Ende des Kapitels wird der Begriff des Clusterings erläutert und die Prinzipien des Hierarchischen und Partitionierenden Clusterings genauer ausgeführt.

Zu Beginn des Kapitels drei wird der IFS-Flow beschrieben. Danach werden die Modelle des Spezifikationsgraphen und des Kommunikationsgraphen vorgestellt. Es folgt eine Übersicht zur Funktion des IFD-Mappings. Abschließend wird eine Erläuterung zum Interface Block und seinen strukturellen Komponenten gegeben.

Im Kapitel vier werden zunächst die Auswirkungen verschiedener Systemtopologien auf die benötigten Ressourcen der Systemarchitektur beschrieben. Für die automatische Verteilung werden Kostenfunktionen und Strategien vorgestellt, die bei der Verteilung berücksichtigt werden. Nach Erläuterung der Bedeutung von Clustering für die Verteilung von IFBs, wird das Verfahren zur Verteilung von IFBs selbst vorgestellt.

Kapitel fünf formuliert die Anforderungen für die Verwendung von Dedicated Protocols. Darauf folgt die Definition des Merged Protocols als mögliche Realisierung eines Dedicated Protocols. Das Konzept des Merged Protocols erfordert die Zerlegung eines Protokoll-Zustandsautomaten in seine Grundblöcke. Diese Zerlegung wird von einem Protokoll-Parser vorgenommen, der im folgenden Abschnitt erläutert wird. Am Ende des Kapitels wird der Algorithmus für die Berechnung eines Merged Protocols angegeben und die Auswirkungen der Verwendung eines Merged Protocols auf die Kommunikationsstruktur beschrieben.

Im letzten Kapitel sechs wird ein Überblick über die vorgestellten Verfahren mit einem Ausblick auf mögliche Erweiterungen gegeben.

2 Grundlagen

2.1 Protokolle

In dieser Studienarbeit wird ein Verfahren erweitert, mit dem die automatische Synthese von Schnittstellen und Protokollen ermöglicht wird, der *Interface Synthesis-Design-Flow* (*IFS-Design-Flow*). Im IFS-Design-Flow sind Protokolle als Teil einer Schnittstellenbeschreibung definiert. In den Protokoll-Beschreibungen wird das Verhalten von Protokollen in Form eines endlichen Zustandsautomaten beschrieben (Finite State Machine = FSM). Die Beschreibung des Verhalten von Protokollen durch eine FSM und die verwendete Definition der FSM wird im Weiteren beschrieben.

Ein Protokoll definiert das Verhalten einer Menge von gerichteten Verbindungen. Die Verbindungen werden durch so genannte Protocol Pins repräsentiert. Das Verhalten der Signale auf diesen Verbindungen und die Signalausgabe werden durch eine FSM beschrieben. In Protokollen ist das Verhalten durch eine Abfolge von Zuständen, die Signalausgabe durch eine entsprechende Ausgabe in einem Zustand modelliert. Die verwendete FSM entspricht einem Moore Automat.

Definition: *Protocol – FSM* = $\{S, X, Y, \delta, \lambda\}$

- S: Zustände
- X: Eingabe ($\Rightarrow$ Externe Signale für Transitionsbedingungen)
- Y: Ausgabe ($\Rightarrow$ Definiert den Wert eines Signals an einem Protocol Pin)
- δ : Zustandsübergangsfunktion: $(S \times (X, (Y|Y = IC))) \rightarrow S$
- λ : Ausgabefunktion: $S \rightarrow Y$

2.2 Graphentheorie

Zum Beschreiben des Verhaltens von Protokollen werden Protokoll-FSMs verwendet. Diese können als gerichtete Graphen visualisiert werden. In späteren Abschnitten dieser Studienarbeit werden Definitionen für Strukturen verwendet, welche sich am besten mit Begriffen aus der Graphentheorie beschreiben lassen.

Ein Graph ist eine Visualisierung von Objekten und deren Beziehungen untereinander. Die Objekte werden im Graphen durch Knoten repräsentiert. Die Beziehungen zwischen den Objekten werden durch Kanten dargestellt. Im folgenden Abschnitt wird die, in dieser Arbeit verwendete, Definition von Graphen und ihren Eigenschaften gemäß [Har69] vorgestellt.

Ein Graph $G = (V, E)$ besteht aus:

- V : eine Menge von Knoten (Vertices) $V = \{v_1, v_2, \dots, v_n\}$
- E : einer Menge von Kanten (Edges) mit $\forall e_i \in E : e = (v_i, v_j)$

In einem ungerichteten Graph ist jede Kante $e = (v_1, v_2)$ ein ungeordnetes Paar. Bei einem gerichteten Graphen ist jede Kante $e = (v_1, v_2)$ ein geordnetes Paar. Wenn $e_j = (v_1, v_2)$ und $e_k = (v_2, v_1)$ dann gilt $e_j \neq e_k$. Des Weiteren sei die Menge der Knoten eines Graphen G mit $V(G)$ bezeichnet. Die Menge der Kanten eines Graphen G sei mit $E(G)$ bezeichnet.

Pfad: Ein Pfad in einem Graphen ist eine Folge von Kanten. Für einen Pfad p in einem Graphen G von v nach w mit $v, w \in V$ ist eine Abfolge von Kanten $p = (e_1, e_2, \dots, e_n)$, wenn gilt $v \in e_1, e_i \cap e_{i+1} \neq \emptyset$ und $w \in e_n$.

Pfadlänge: Die Länge eines Pfades entspricht der Anzahl der Kanten auf dem Pfad.

Kreis: Ein Kreis ist ein Pfad in einem Graphen, bei dem der erste und letzte Knoten gleich sind. Da die sich die Abfolge der Knoten in einem Kreis wiederholt, wird ein Kreis auch als *Zyklus* bezeichnet.

Grad: Der Grad $d(v)$ eines Knoten v ist die Anzahl der Kanten e_i für die gilt: $v \in e_i$

Disjunkte Graphen: Zwei Graphen sind disjunkt wenn sie keine gemeinsamen Knoten besitzen. G_1 ist disjunkt zu G_2 wenn gilt: $V(G_1) \cup V(G_2) = \emptyset$

Verbindung: Auf Graphen lassen sich verschiedenste Operationen definieren wie z.B. das Komplement, die Vereinigung oder Differenz von Graphen. Eine der möglichen Operationen für die Kombination von Graphen ist die Verbindung (siehe Abbildung 2.1). Der Graph G sei das Ergebnis der Verbindung zweier disjunkter Graphen G_1 und G_2 . Dann ist G definiert durch:

$$\begin{aligned} G &= G_1 + G_2 \text{ mit} \\ V(G) &= V(G_1) \cup V(G_2) \\ E(G) &= E(G_1) \cup E(G_2) \cup \{v_1 v_2 \mid v_i \in V(G_i), i = 1, 2\} \end{aligned}$$

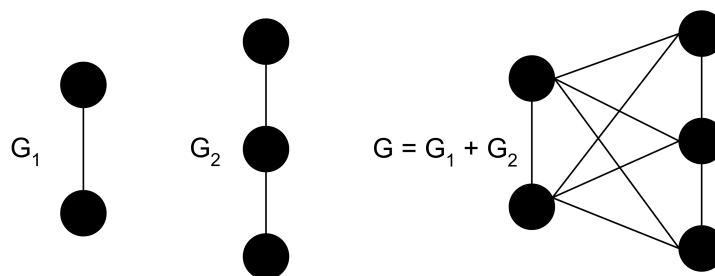


Abbildung 2.1: Verbindung von zwei Graphen

2.3 Protokoll Grundblöcke

Protokolle beinhalten neben den Signalen für die zu übertragenden Daten (*Nutzdaten*) weitere Signale zur Steuerung des Ablaufs oder Sicherung der Datenübertragung (*Redundanz*). Je nach Art des Protokolls kann man daher den Ablauf des Protokolls in mehrere Abschnitte aufteilen, die entweder der Steuerung des Ablaufs oder der Datenübertragung dienen. Diese werden im weiteren Verlauf als *Grundblöcke* bezeichnet. Die Zerlegung eines Protokolls in seine Grundblöcke ist an mehreren Stellen des IFS-Flow relevant.

Analog zur Definition von Grundblöcken im Software-Design können Grundblöcke auch für Protokolle definiert werden. Gegeben sei der Graph einer Protokoll-FSM. Die Zustände des Automaten werden durch Knoten, die Transitionen durch die Kanten eines gerichteten Graphen repräsentiert.

Definition Grundblock (Basic Block = BB): *Ein Grundblock ist eine geordnete Menge von Zuständen, die auf einem kreisfreien Pfad maximaler Länge in einem Protokoll Graphen liegen. Auf diesem Pfad dürfen nur der erste und der letzte Knoten einen Grad größer eins haben. Der Knoten welcher dem Startzustand der Protokoll-FSM entspricht, darf nur als erster Knoten in einem solchen Pfad enthalten sein.*

Ein Grundblock beginnt, wenn eine der folgenden Bedingungen zutrifft:

1. Der Startzustand einer Protokoll-FSM ist immer der Anfang eines Grundblocks.
2. Zustände die nach einem Zustand mit mehreren Folgezuständen liegen (*Gabelung, fork of control*) sind immer der Anfang von Grundblöcken.
3. Zustände welche Nachfolger von mehreren Zuständen sind (*Zusammenführung, merge of control*) sind ebenfalls Anfangszustände eines Grundblocks.

Nur am Anfang oder am Ende eines BBs kann es daher zu einer Verzweigung im Ablauf eines Protokolls kommen. Im Grundblock selbst gibt es keine Verzweigung. Datenpakete sollten in syntaktisch korrekten Protokollen immer vollständig übertragen werden. Die vorzeitige Beendigung der Übertragung eines Datenpaketes sollte im Normalfall nicht im Ablauf eines Protokolls vorkommen. Für den IFS-Flow wird vorausgesetzt, dass Datenpakete nicht über die Grenzen eines Grundblocks hinausgehen.

Wenn der Graph eines Protokollautomaten nur durch Grundblöcke und Transitionen zwischen diesen Grundblöcken dargestellt wird, so bezeichnen wir das im Weiteren als Kontrollflussgraph (Control Flow Graph = CFG).

Definition Kontrollflussgraph (CFG):

Ein Kontrollflußgraph $CFG = \{V; E\}$ besteht aus

$V = \{\text{Grundblöcke}\}$

$E \subseteq V \times V$, eine Menge von Kanten (= *Transitionen*) zwischen Grundblöcken.

Abbildung 2.2 zeigt einen CFG der aus 4 Grundblöcken BB_1 bis BB_4 besteht. Der CFG wurde zu einem Protokoll mit den Zuständen S_1 bis S_5 erstellt.

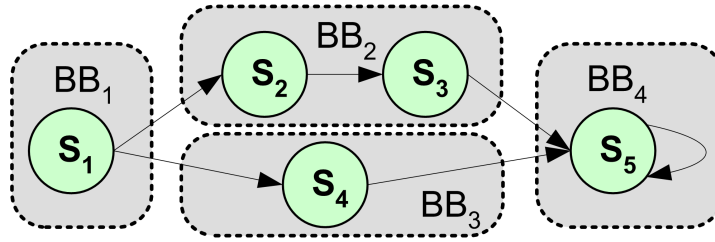


Abbildung 2.2: Beispiel für einen Control Flow Graph

2.4 Clustering

Im folgenden Abschnitt wird der Begriff des Clusterings erläutert. Bei dem Clustering werden gleichartige Objekte in Gruppen (Cluster) zusammengefasst. Clustering wird in sehr unterschiedlichen Bereichen angewendet. Beispiele finden sich unter anderem in der Biologie, Medizin oder Mathematik. Viele Entscheidungs-, Optimierungs- und Lernprobleme können durch den Einsatz von Clusteringalgorithmen gelöst werden.

Aufgrund der vielen Anwendungsbereiche gibt es zahlreiche unterschiedliche Algorithmen, welche jeweils eine speziell auf das Problem zugeschnittene Form des Clusterings realisieren. Welche Objekte in einem Cluster zusammengefasst werden, wird durch die Ähnlichkeit der Objekte zueinander festgelegt. Die Ähnlichkeit kann durch ein oder mehrere Attribute eines Objekts bestimmt werden. Man unterscheidet zwischen quantitativen und qualitativen Attributen. Ähnlichkeit kann durch eine beliebige Größe definiert werden.

Oft wird die Distanz zwischen Objekten für das Bestimmen der Ähnlichkeit verwendet. Je kleiner die Distanz, desto höher die Ähnlichkeit. Man spricht dann von einer Distanzfunktion. Die Distanz zwischen zwei Elementen x und y der Menge D , wird mit $d(x, y)$ bezeichnet. Für die Distanzfunktion $d(x, y)$ müssen drei Bedingungen erfüllt sein:

$$\forall x, y \in D : d(x, y) \in \mathbb{R}^+ \quad (2.1)$$

$$d(x, y) = 0 \Leftrightarrow x = y \quad (2.2)$$

$$d(x, y) = d(y, x) \quad (2.3)$$

Allgemeiner betrachtet kann Clustering als eine spezielle Form der Klassifizierung gesehen werden. In [AKJ88] wird Clustering daher folgendermaßen beschrieben: “A clustering is a type of classification imposed on a finite set of objects“. Der Ausgang einer Klassifikation ist eine Menge von Klassen und Objekte deren Zugehörigkeit zu den einzelnen Klassen bekannt ist. Diese bekannte Menge von Objekten wird als Trainingsmenge bezeichnet. Mit Hilfe der Informationen dieser Trainingsmenge wird dann versucht neue, unbekannte Objekte aufgrund ihrer Eigenschaften in Klassen einzuteilen. Da die Objekte und die Klassen der Trainingsmenge bereits bekannt sein müssen, wird diese Art der Klassifikation als *überwachtes Lernen* bezeichnet. Dem gegenüber steht die als *unüberwachtes Lernen* bezeichnete Art der Klassifizierung, das Clustering (siehe Abbildung 2.3).

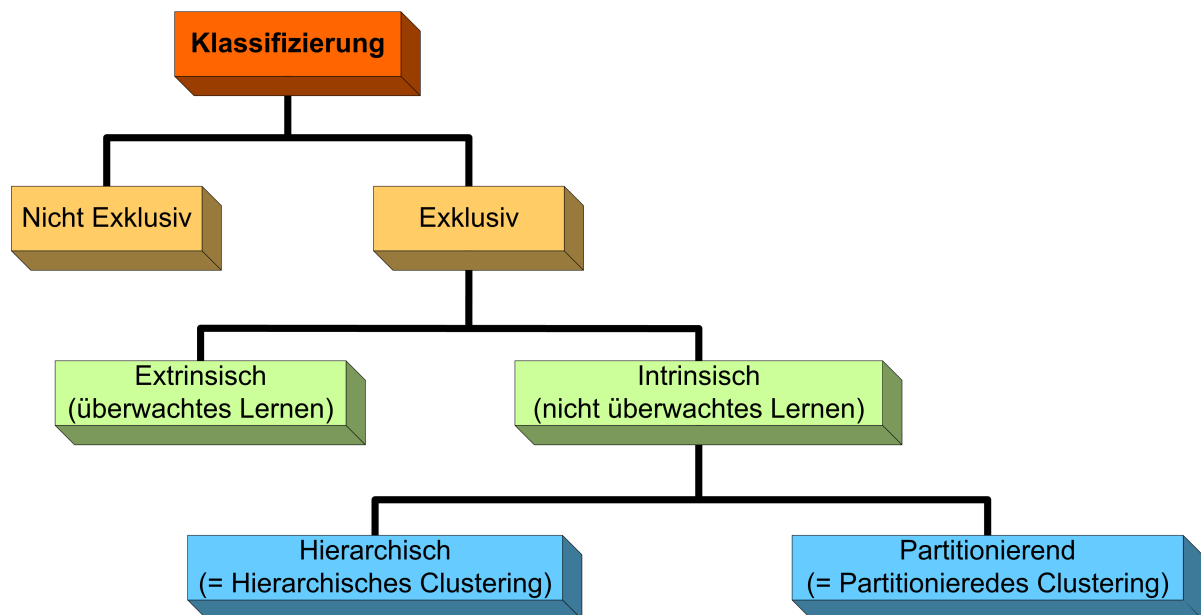


Abbildung 2.3: Zusammenhang von Klassifikationsproblemen nach [GNL67]

Dieses kann wiederum in verschiedene Unterarten aufgeteilt werden, was sich nach der Art des Verfahrens richtet. Gemäß der Aufteilung nach [GNL67] sollen im folgenden die beiden Hauptarten der Verfahren vorgestellt werden, das hierarchische und das partitionierende Clustering.

2.4.1 Hierarchisches Clustering

Bei dem hierarchischen Clustering wird eine Menge von Objekten rekursiv in eine Hierarchie von Clustern aufgeteilt, bis jeder Cluster genau ein Objekt enthält. Die Ergebnismenge der Cluster muss nicht vollständig disjunkt sein. Cluster auf tieferen Ebenen sind Untermengen von Clustern auf höheren Ebenen. Die Hierarchie der Ergebnismengen kann in einem Baumdiagramm (Dendrogramm, siehe Abbildung 2.4) dargestellt werden. Die Wurzel des Baums entspricht der gesamten Menge der Objekte. Die Blätter entsprechen jeweils einem einzelnen Objekt. Die Darstellung der Hierarchie in einem Dendrogramm liefert eine schnelle Übersicht weiterer Eigenschaften der Objekte und Cluster zueinander. Je weiter eine Ebene des Dendrogramms von der Wurzel entfernt liegt, desto größer ist die Nähe der einzelnen Objekte in einem Cluster. Mit steigendem Abstand zwischen Clustern auf einer Ebene steigt ihre Unterschiedlichkeit.

Man kann nur einen Teil der Mengen, welche ein hierarchischer Clusteringalgorithmus liefert, als Ergebnis betrachten. Wenn man nur Cluster auf einer bestimmten Ebene betrachtet erhält man disjunkte Cluster. Wählt man tiefere Ebenen erhält man eine feinere Einteilung mit mehr Clustern als in einer höheren Ebene. Algorithmen welche ein hierarchisches Clustering erzeugen benötigen eine Distanzfunktion um zwei einzelne Objekte zusammenzufassen und eine Distanzfunktion für das Zusammenfassen von Mengen von

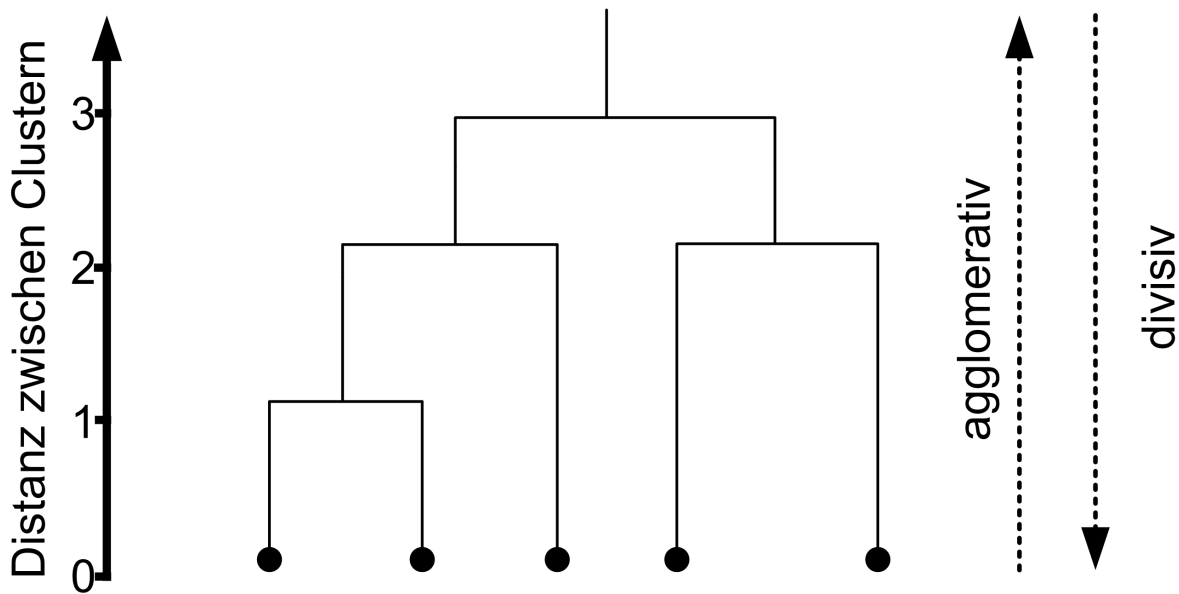


Abbildung 2.4: Beispiel für ein Dendrogramm

Objekten (Clustern). Im Folgenden werden einige verschiedene Arten von Verfahren zur Durchführung von hierarchischen Clustering beschrieben.

Agglomeratives Clustering

Bei dem agglomerativen Verfahren wird ein bottom-up Ansatz verwendet, um die Hierarchie der Cluster aufzubauen. Als Ausgangslage verwendet man Cluster mit genau einem Element (unterste Ebene). Die nächste Ebene der Hierarchie wird durch Zusammenfassen von mehreren Clustern auf unterster Ebene erzeugt. Dieser Vorgang wird so lange wiederholt, bis man auf der obersten Ebene der Hierarchie angekommen ist. Auf dieser Ebene sind alle Objekte in einem Cluster zusammengefasst.

Diversives Clustering

Diversive Verfahren verwenden einen top-down Ansatz und arbeiten genau in umgekehrter Reihenfolge zu agglomerativen Verfahren. Alle Objekte werden zuerst in einem Cluster zusammengefasst. Dieses wird in kleinere Cluster aufgeteilt, welche dann die nächste Ebene der Hierarchie bilden.

Eine Möglichkeit besteht darin ein Objekt mit der größten Distanz zu den übrigen Objekten zu suchen. Aus diesem Objekt wird ein neuer Cluster gebildet. Dann wird die durchschnittliche Distanz zwischen einem Objekt des ursprünglichen Clusters und allen Objekten des neu gebildeten Cluster berechnet. Von diesem Wert wird die durchschnittliche Distanz des gewählten Objektes zu allen anderen Objekten im gleichem Cluster abgezogen. Der so erhaltene Wert wird für jedes Objekt im ursprünglichen Cluster ermittelt. Das Objekt bei dem so der höchste Wert ermittelt wurde, ist den Objekten im

neuen Cluster ähnlicher als den Objekten im eigenen Cluster.

Dieses Objekt wird in den neuen Cluster verschoben. Dieser Vorgang wird so lange wiederholt, bis alle errechneten Werte negativ sind. Dann ist die Menge der Objekte für das neue Cluster vollständig. Die so erhaltenen Cluster können nach der gleichen Methode weiter aufgeteilt werden, bis alle Cluster der untersten Ebene genau ein Element enthalten.

Inkrementelles Clustering

Bei inkrementellen Verfahren wird jeweils ein Objekt betrachtet und in die Cluster der Hierarchie eingefügt. Dabei wird die Hierarchie und damit der Graph des Dendogramm inkrementell aufgebaut (siehe auch [ZRL96]).

Ausgangslage ist ein Cluster mit einem Element. Dieser Cluster bildet die Wurzel der Hierarchie und des Dendogramm-Graphen mit dem man die Hierarchie darstellen kann. In die Cluster der Hierarchie werden weitere Objekte nach festen Kriterien eingefügt. Erfüllt ein Objekt die Bedingung für die vorhandenen Cluster nicht, werden weitere Cluster in die Hierarchie eingefügt. Die Knoten und Blätter des Dendogrammes entstehen so inkrementell.

2.4.2 Partitionierendes Clustering

Verfahren, die ein partitionierendes Clustering durchführen, liefern nach [AKJ88] zu n Objekten aus einem d -dimensionalen Vektorraum eine Aufteilung der n Objekte in k Cluster, so dass die Objekte in einem Cluster zueinander ähnlicher sind als zu Objekten in einem anderen Cluster.

Der Wert für k kann je nach Verfahren festgelegt oder durch das Clustering errechnet werden. Ein partitionierendes Verfahren teilt also eine Menge von Objekten so in disjunkte Cluster ein, dass jedes Objekt in genau einem Cluster ist und ein Cluster aus mindestens einem Objekt besteht.

Da die Cluster, welche das Ergebnis eines partitionierenden Verfahren darstellen, alle disjunkt sind, gibt es keine Hierarchie der Cluster untereinander. Für das Partitionierende Clustering wird häufig eine Bewertungsfunktion verwendet um die Qualität eines Clusterings auszudrücken. Mit Hilfe der Bewertungsfunktion wird versucht eine optimale Lösung für ein Clusteringproblem zu finden. Ist die Bewertungsfunktion eine Kostenfunktion, so wird ein optimales Clustering gefunden, indem die Kostenfunktion minimiert wird.

Um die richtige Aufteilung der Objekte auf die Cluster zu finden, könnte man alle möglichen Kombinationen ausprobieren. Die Anzahl der möglichen Kombinationen ist ein entscheidender Faktor für den Aufwand, der bei diesem Vorgehen entstehen würde.

Definition: Es sei M eine Menge, es sei k eine natürliche Zahl.

Eine Menge $\{M_1, M_2, \dots, M_k\}$ von k paarweise disjunkten nicht leeren Teilmengen von M mit $M = \bigcup_{i=1}^k M_i$ heißt eine k -Partition von M .

Partitionierendes Clustering von n Objekten auf k Cluster bedeutet im mathematischen Sinne eine k -Partition von n zu bilden.

Die Anzahl der möglichen k -Partitionen einer n -elementigen Menge wird mit $S_{n,k}$ bezeichnet [KHK96]. Wenn n und k natürliche Zahlen sind, dann gilt für $S_{n,k}$:

$$S_{n,k} = S_{n-1,k-1} + k \cdot S_{n-1,k}$$

$$S_{n,k} = \frac{1}{k!} \sum_{i=1}^k (-1)^{k-i} \binom{k}{i} (i)^n$$

Die Lösungen für $S(n,k)$ sind die so genannten *Stirling Zahlen zweiter Ordnung*¹. Bei 19 Elementen und 4 Partitionen ist $S(n,k) = 11.259.666.000$. Der Lösungsraum ist auch bei relativ kleinen Werten für n und k bereits sehr groß. Alle möglichen Lösungen zu errechnen und zu Bewerten ist daher oft nicht durchführbar.

Die meisten Verfahren verfolgen daher einen iterativen Ansatz, bei dem versucht wird, ein initiales Clustering zu optimieren. Die Qualität der gefundenen Lösung kann dabei sehr stark von dem gewählten initialen Clustering abhängig sein. Es besteht die Gefahr, dass die gefundene Lösung nur aufgrund eines lokalen Extremum der Bewertungsfunktion akzeptiert wird. Durch wiederholtes Durchführen des Clusterings mit unterschiedlichen initialen Clusterings kann man mehrere Lösungen erzeugen. Nimmt man von diesen Lösungen die optimale, so kann die Wahrscheinlichkeit, eine Lösung aufgrund eines lokalen Extremum der Bewertungsfunktion zu akzeptieren, verringert werden.

Das Erstellen eines guten Clustering Algorithmus wird dadurch erschwert, dass es meist keine eindeutig Lösung für die Optimierung einer Bewertungsfunktion gibt. Die Optimierung der Bewertungsfunktion ist dann eine systematische Suche im Lösungsraum. Alle diese Faktoren führen zu einer immensen Anzahl von Ansätzen für Partitionierendes Clustering.

¹James Stirling (22. April 1692 - 1770)

3 Interface Synthesis Design Flow

Alle im Rahmen dieser Arbeit erstellten Verfahren stehen im Zusammenhang mit dem Ansatz des *Interface Synthesis Design Flow*. Dieser wird in dem folgenden Kapitel, zusammen mit den Modellen des Spezifikations- und Kommunikationsgraphen, genauer erläutert.

3.1 Motivation für automatisiertes Design

Das Gebiet der eingebetteten Systeme entwickelt sich rasant hinsichtlich Komplexität und Vielfalt. Die meisten eingebetteten Systeme bestehen aus verteilten aber hochgradig untereinander verbundenen Anwendungen. Zuverlässigkeit und Effektivität sind wichtige Eigenschaften für das Design und die Implementation eines Kommunikationssystems. Die Verwendung von standardisierten Schnittstellen und Kommunikationsprotokollen sind wichtige Techniken, um die Zuverlässigkeit und Effektivität sicher zu stellen.

Die erhöhte Komplexität und die Notwendigkeit von kurzen Entwicklungszyklen zwingen den Designer förmlich die Wiederverwendung von Intellectual Properties (IP) zu berücksichtigen. Da die IPs von einer Vielzahl von verschiedenen Herstellern angeboten werden, ist man zwangsläufig mit verschiedensten inkompatiblen Schnittstellen und nicht konformen Protokollen konfrontiert. Daher müssen durch den Designer eines Systems Adapter erstellt werden, mit deren Hilfe IPs eingebunden werden können. Ein vollständiges Verständnis aller beteiligten Schnittstellen ist für diesen Prozess erforderlich. Die Einbindung bereits vorhandener IPs ist einer der zentralen und auch fehleranfälligen Aspekte in der Entwicklung von neuen eingebetteten Systemen [PRSV98].

Der so genannte *Interface Synthesis Design Flow* (IFS-Flow) ist eine Methode zum modellieren und automatischen Synthetisieren von rekonfigurierbaren Schnittstellen in eingebetteten Systemen (siehe Abbildung 3.2). Die Integration und flexible Wiederverwendung von IPs war ein integraler Bestandteil bei der Entwicklung des IFS-Flow. Daher bietet der IFS-Flow die Möglichkeit einer automatisierten Synthese von Schnittstellen als Adapter für inkompatible IPs [Ihm04].

Bei diesem Verfahren ist das vollständige Verständnis der einzubinden IP nicht mehr erforderlich. Das bedeutet auch, dass die innere Funktionalität einer IP geheim gehalten werden kann (*black box*). Diese Eigenschaften ermöglichen einen flexiblen und schnellen Entwurf von Prototypen. Für die automatisierte Integration von IPs muss nur die *Schnittstellenbeschreibung* (Interface Description, IFD) zur Verfügung stehen. Die *IFS Systemarchitektur* wurde für die Entwicklung komplexer Kommunikationsprototypen erdacht. Wie in Abbildung 3.1 dargestellt, wird eine hierarchische Struktur aus folgenden *System-Komponenten* zusammengestellt: *System*, *Board*, *Chip*, *Task* und *Medium*.

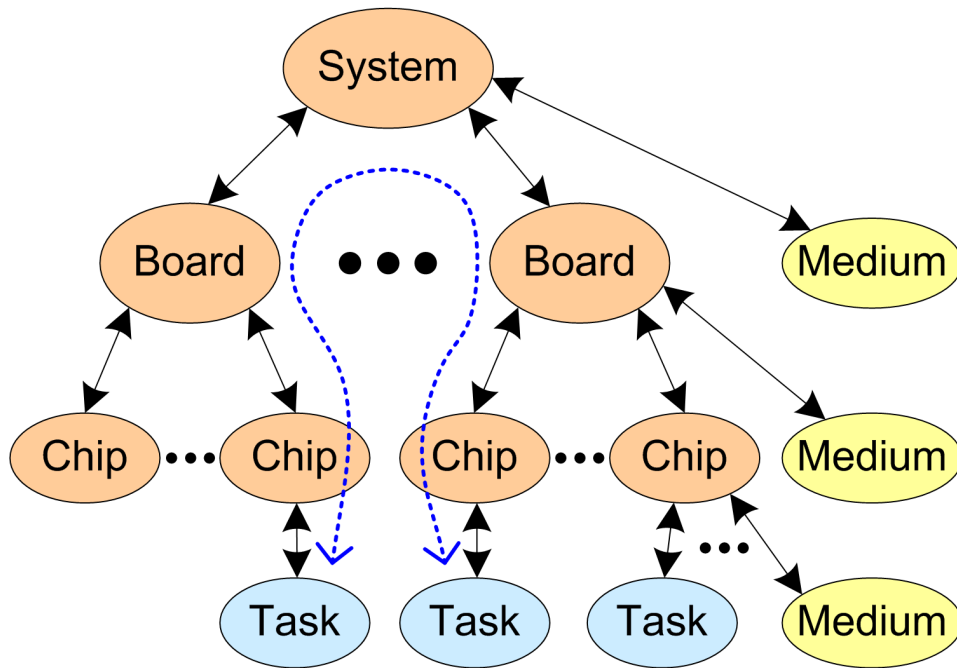


Abbildung 3.1: IFS System-Architektur

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, aber nur die Kommunikations-Komponenten besitzen Verhaltensbeschreibungen in Form von Protokoll-Zustands-Automaten. Zwei kompatible Tasks (oder Medien) können über die vorhandenen Verbindungen innerhalb der System-Architektur kommunizieren. Inkompatible Kommunikations-Komponenten hingegen werden über einen so genannten Interface Block (IFB) verbunden.

Grundlage für den Entwurfsprozess nach dem IFS-Flow bildet die abstrakte Beschreibung der Systemarchitektur. Sie wird als Menge von IPs beschrieben. Das Format für diese Beschreibungen ist das so genannte Interface Synthesis Format. Dieses Format beschreibt als XML-Schema die physikalische Struktur, die Eigenschaften und die Protokolle aller Komponenten. Mit diesen Informationen wird die automatische Synthese ermöglicht.

Für die Synthese werden zwei Arten von Beschreibungen benötigt: TPD und IFD. Die Beschreibung der Ziellplattform wird als *Target Plattform Descriptions (TPD)* bezeichnet. Sie beschreibt die Eigenschaften der Chips als Ausführungsplattform. Die Schnittstellen von Komponenten werden durch die *Interface Descriptions (IFD)* beschrieben. Die Beschreibung eines Systems und seiner Komponenten im IFS-Format kann mit Hilfe des Java-Werkzeuges *IFS-Editor* erzeugt werden.

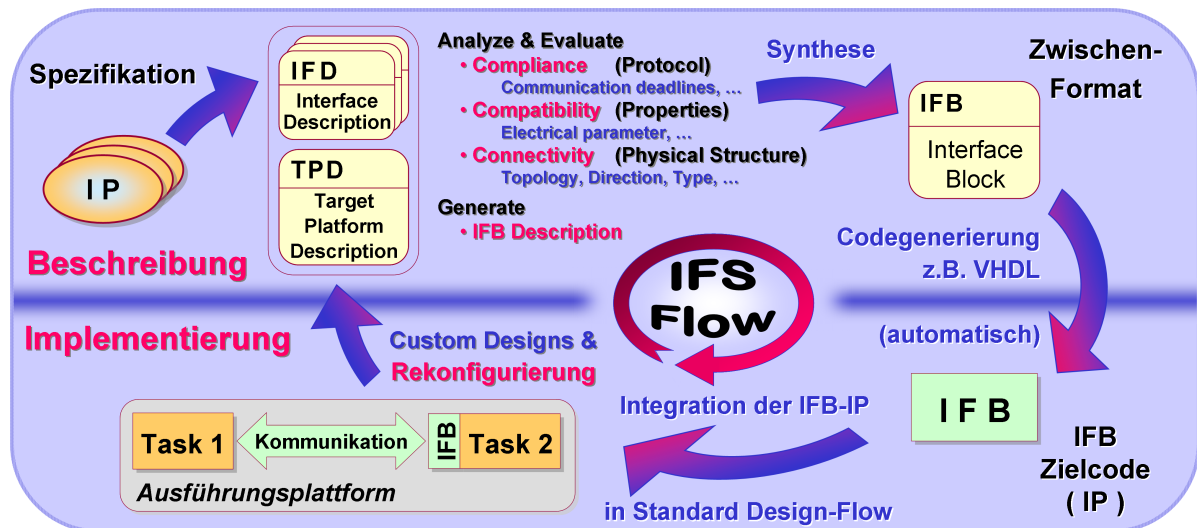


Abbildung 3.2: IFS-Flow [Ihm04]

Ist ein vollständiges System mit den gewünschten IPs im IFS-Format vorhanden, wählt der Entwickler die Schnittstellenbeschreibungen der Task- und Medienschnittstellen aus, welche verbunden werden sollen. Es folgt eine Analysephase, in welcher die Übereinstimmung, Kompatibilität und die Konnektivität der ausgewählten IFDs überprüft wird. Wenn alle notwendigen Bedingungen erfüllt sind, kann eine einfache, direkte Verbindung erstellt werden. Andernfalls, wenn die Protokolle inkompatibel sind, wird ein IFB synthetisiert.

Die Synthese teilt sich in zwei Phasen. In der ersten Phase wird eine abstrakte IFB Instanz in einem Zwischenformat erstellt. Dieses Zwischenformat kann als XML-Datei gespeichert, exportiert und importiert werden. In der zweiten Phase wird der endgültige Code der IFB Implementierung automatisch generiert. Die Zielsprache hängt davon ab, ob eine Hardware- oder Softwarelösung erstellt werden soll. Zur Zeit wird VHDL für die Synthese von Hardware unterstützt. Zum Schluss wird die so erzeugte IFB Instanz in die bereits existierende Implementierung der modellierten Systemarchitektur integriert.

Der IFB-Flow setzt dabei das Vorhandensein der Implementierung einer jeden beschriebenen IP voraus. Die in dieser Arbeit vorgestellte Erweiterung des IFS-Flows ermöglicht eine automatisierte und optimierte Erstellung aller benötigten Schnittstellen in einem Design Zyklus. Dafür wird das Erstellen und automatische Verteilen von mehreren Multi-Task IFBs in den Synthese Schritten ermöglicht. Zusätzlich werden Dedicated Protocols verwendet um den Kommunikationsoverhead zu reduzieren und strukturelle Engpässe im Kommunikations-System zu überwinden. Beide Erweiterungen des IFS-Flows ermöglichen das Beschleunigte und Optimierte Erstellen von Kommunikationssystemen.

3.2 Spezifikationsgraph

Ein Ansatz zur Modellierung auf Systemebene und Hardware / Software Codesign unter Berücksichtigung von Schnittstellen ist in [Tei97] gegeben. Das Designproblem wird durch einen Spezifikationsgraphen beschrieben. Dieser setzt sich aus einem Problem- und einem Architekturgraphen zusammen. Eine optimale Abbildung des Problemgraphen auf den Architekturgraphen ist die gesuchte Lösung.

Ein Problemgraph setzt sich aus Task-Knoten und Kommunikationsknoten zusammen, die zum Verbinden der Tasks genutzt werden. Die Task-Knoten müssen auf die Architekturkomponenten abgebildet werden, welche die ausführenden Plattformen innerhalb des Architekturgraphen darstellen.

Die Kommunikationsknoten hingegen müssen auf die Architekturkomponenten abgebildet werden, welche die Kommunikationskanäle darstellen. Dies ist ein kritischer Aspekt, da der Entwickler sicherstellen muss, dass die verwendeten Kommunikationskanäle zu den angebundenen Tasks, in Hinblick auf das Medium, das Protokoll und den benötigten Datendurchsatz passen.

3.3 Kommunikationsgraph

Die automatische Interface Synthese in Verbindung mit konfigurierbarer Hardware ermöglicht die Definition von Full Custom Interfaces welche auf der Zielplattform implementiert werden müssen. Full Custom Interfaces entstehen dadurch, dass nur die tatsächlich benötigten Teile der Schnittstellen in der Synthese beachtet werden. Welche Teile das sind, kann im IFD-Mapping spezifiziert werden. Der IFB ist als Kommunikationspartner für die anderen Komponenten transparent. Daher reduzieren sich die Kommunikationskanäle des Architekturgraphen auf einfache Verbindungen. Dies ermöglicht einen hohen Grad an Flexibilität in der Konstruktion von Kommunikationssystemen.

Der in Abbildung 3.3 dargestellte *Kommunikationsgraph* wurde durch die Verbindung der *IFS Systemarchitektur* mit dem Spezifikationsgraph Modell erzeugt. Der Architekturgraph leitet sich von den *Architekturkomponenten* der System-Architektur ab (siehe auch Abbildung 3.1).

Alle Task-Knoten (hier: $T_1, \dots, T_5$) müssen auf die vorhandenen Chips abgebildet werden. Im Unterschied zum Spezifikationsgraph werden die Kommunikationsknoten auf IFBs abgebildet. Mindestens ein (Multi-Task) IFB wird benötigt, um eine Menge von inkompatiblen Tasks zu verbinden. Die IFBs werden zur Design-Zeit angelegt und danach synthetisiert, wie im IFS-Flow beschrieben. Im Unterschied zum Spezifikationsgraphen ist der IFB, welcher als Umsetzung eines Kommunikationsknoten gesehen werden kann, an die Implementations-Plattform (Chip) gebunden.

3.4 Das IFD-Mapping

Jeder *Kommunikationsknoten* (hier: $C_1, \dots, C_4$) repräsentiert ein *IFD-Mapping*. Ein IFD-Mapping besteht aus Mapping Equations und spezifiziert die Umsetzung der Informa-

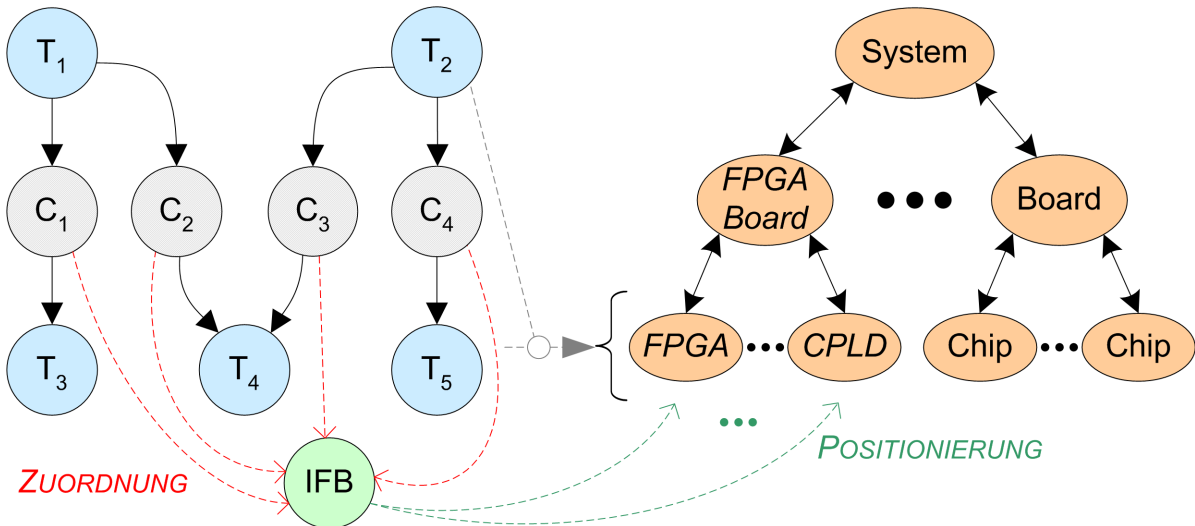


Abbildung 3.3: Kommunikationsgraph

tionen von eingehenden zu ausgehenden Daten innerhalb eines IFB. Die *IFD-Mapping Sprache* wurde definiert, um Mapping Equations *MapEqn* der folgenden Form zu beschreiben:

$$MapEqn : data_{Out} \leq f_{Map}(data_{In,1}, \dots, data_{In,k});$$

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

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

Die Datenpaket, welche sich aus einem Protokoll extrahieren lassen, können als Matrix repräsentiert werden. Daher wurde die Notation für das Umsortieren von Bits oder das Zuweisen von konstanten Werten an die Matrix Notation von Matlab angelehnt.

$$\text{z.B.: } P_{Out}[1..2] \leq P_{In}[2..1];$$

Für komplexere Funktionen, wie verknüpfte boolesche Funktionen oder die Verwendung einer FSM, wurden weitere Konstrukte in die IFD-Mapping Sprache aufgenommen.

$$P_{Out} \leq \text{if } (P_{In}[0] = '0') \{P_{In}\} \text{ else } \{ "0110" \};$$

Um mehrere IFD-Mappings verschiedener Tasks in einem IFB zu bearbeiten, kann ein *Multi-Task IFB* konstruiert werden. Das Zuordnen (Binding) eines IFD-Mappings zu einem IFB bedeutet das Implementieren des IFD-Mappings im IFB. Um das Verteilen von IFD-Mappings, unter Berücksichtigung einer gegebenen System-Architektur, weiter zu erörtern ist ein Grundverständnis der der internen IFB Struktur erforderlich.

3.5 Der Aufbau des Interface Blocks (IFB)

Ein Interface Block (IFB) ist ein Adapter Modul, das durch den IFS-Flow dynamisch erzeugt wird. Der IFB dient als Schnittstelle zwischen Kommunikationskomponenten. Durch die so erzeugte Schnittstelle können IPs integriert werden, ohne Änderungen an den IPs selbst vornehmen zu müssen. Der IFB erlaubt dabei in Zusammenspiel mit dem IFD-Mapping auch das Einbinden inkompatibler IPs, da das IFD-Mapping Funktionen bietet, welche über einfaches Abbilden von Pins hinausgehen.

Für das Erzeugen eines IFB werden die Schnittstellenbeschreibungen der verbundenen Tasks [Ihm03b] zusammen mit dem IFD-Mapping ausgewertet und die benötigten Komponenten modular synthetisiert. Ein IFB realisiert die Umsetzung der Mapping Equations. Mapping Equations ermöglichen die Umwandlung von Datenpaketen die zwischen inkompatiblen Protokollen übertragen werden. Über einen IFB wird daher das Verbinden von Komponenten mit unterschiedlichen Protokollen ermöglicht.

Das ist einer der größten Vorteile des IFS-Flows. Die Erzeugung von IFBs ist daher ein zentraler Aspekt des IFS-Flows [Ihm03a, Ihm02b, Ihm02a]. Die Kommunikationsstruktur im Inneren eines IFB wird als *IFB-Makrostruktur* bezeichnet. Die IFB-Makrostruktur wird durch eine Reihe von endlichen Zustandsautomaten (*Finit State Machine* = *FSM*) realisiert. Die einzelnen Komponenten dieser Struktur werden im Weiteren detaillierter beschrieben.

3.5.1 Control Unit (CU)

Die Struktur des IFB unterteilt sich in drei Komponenten: *Protocol Handler* (PH), *Sequence Handler* (SH) und *Control Unit* (CU) (siehe Abbildung 3.4).

Die CU regelt die interne Steuerung zwischen den einzelnen Komponenten der Makrostruktur des IFB. Dazu sendet die CU Steuersignale zum PH und SH und wertet deren Statussignale aus. In einem Multi-Task IFB übernimmt die CU auch das Scheduling des Zugriffs einzelner Komponenten auf die IFB internen Ressourcen.

Die CU beinhaltet zwei Scheduler. Einer steuert, welcher PH-Modus gerade Daten empfangen darf. Dazu gibt er einen entsprechenden Bus im IFB frei, über den eingehende Daten von einem PH-Modus in den Speicher für eingehende Daten geschrieben werden. Der andere Scheduler entscheidet welcher PH-Modus Daten versenden darf. Dazu wird ebenfalls ein entsprechender Zugriff auf den Bus verwaltet, der mit dem Speicherbereich für die ausgehenden Daten verbunden ist. Als Scheduling Verfahren können verschiedene Varianten verwendet werden.

Ein Scorebord stellt sicher, dass die Kausalität während der Ausführung des *Eingabe-Verarbeitung-Ausgabe (EVA)* Zyklus erhalten bleibt. Der EVA Zyklus in einem IFB läuft nach dem Prinzip des Pipelining ab.

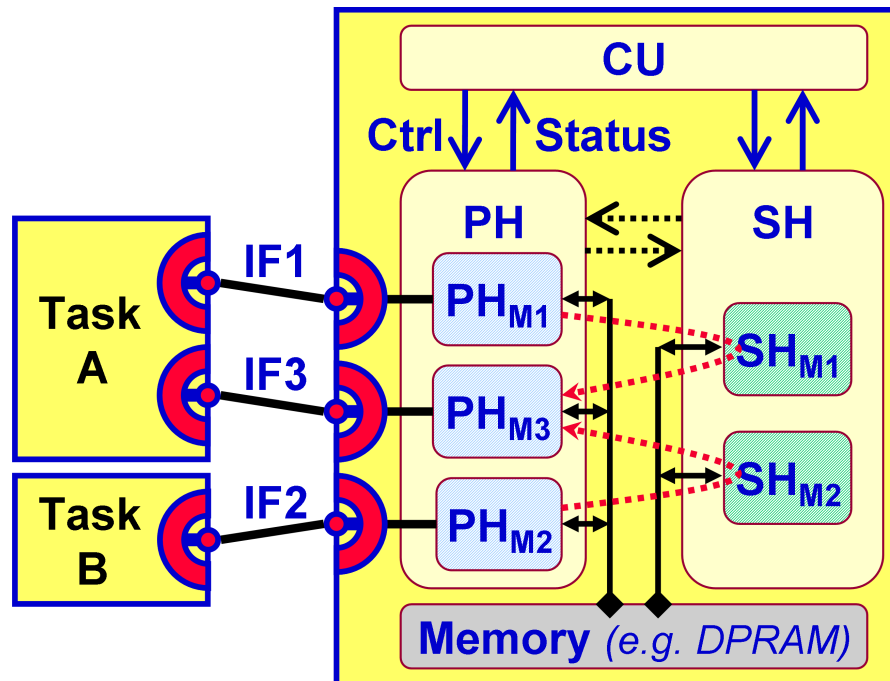


Abbildung 3.4: IFB-Makrostruktur

3.5.2 Protocol Handler (PH)

Sequence und Protocol Handler sind modular aus so genannten Modi aufgebaut. Diese Modi realisieren Funktionen und sind jeweils durch eine FSM beschrieben. Der Protocol Handler besitzt einen Modus pro Kommunikationsverbindung mit einer angebotenen Task. Der PH realisiert eine oder mehrere Schnittstellen zu den Kommunikationspartnern, die über einen IFB verbunden sind. Über diese Schnittstellen laufen die ein- und ausgehenden Daten. Für jede verbundene Schnittstelle einer Task wird ein Protocol Handler Mode PH_M im PH benötigt, der das entsprechende Protokoll der angeschlossenen Task verarbeitet.

Ein PH_M implementiert dazu das komplementäre Protokoll aus einer Task, welches in der dazugehörigen IFD beschrieben ist. Hierfür wird die Protokoll FSM von Task t aus der IFD extrahiert und eine FSM generiert, welche ein komplementäres Protokoll realisiert. Diese FSM wird um zusätzliche Zustände erweitert. Mit diesen Zuständen wird die Interaktion mit der Control Unit und dem Sequence Handler ermöglicht. Ein PH_M übernimmt das Einfügen und Extrahieren von Daten eines Protokolls gemäß der Mapping Equation des Protokolls. Alle nicht im IFD-Mapping vorkommenden Daten

werden ignoriert. Die Rohdaten werden in einem Speicherelement des Sequence Handlers zwischengespeichert.

3.5.3 Sequence Handler (SH)

Der Sequence Handler beinhaltet eine Menge von Modi, welche die Umsetzung des IFD-Mappings realisieren. Für jede Mapping Equation in einem SH existiert eine Sequence Handler Mode (SH_{M_x}). Die Rohdaten im Speicherelement werden durch den SH_M modifiziert. Dazu werden die gespeicherten Daten umsortiert oder modifiziert, so wie es im IFD-Mapping spezifiziert ist. Dabei sind nicht nur strukturelle, sondern auch semantische Änderungen der Daten möglich. Für die Kommunikation zwischen unterschiedlichen Protokollen ist das erforderlich. Die modifizierten Daten werden an einer anderen Stelle im Speicherelement abgelegt. Ist ein Datenpaket vollständig, werden die Daten im ausgehenden Protokoll durch einen zugehörigen PH_M zusammengefügt.

Ein Hauptvorteil des Modulare Aufbaus des IFB, ist das Umsetzen der vollständigen Schnittstellen Funktionalität, ohne dass der Produkt-Automat aller Ursprungs- und Ziel-FSMs erstellt werden muss. Die Verwendung von Modi erlaubt eine einfache Anpassung und Erweiterung eines IFB. Sogar eine Rekonfiguration zur Laufzeit ist möglich [Ihm04].

Die Verwendung von IFD-Mappings in Verbindung mit den SH Modes erlauben eine maximale Flexibilität im Datenaustausch. Ein automatisches IFD-Mapping zwischen den empfangene und gesandten Daten ist unmöglich ohne vollständiges Verständnis der Semantik der ausgetauschten Informationen. Die Verwendung von IPs unterschiedlicher Hersteller unter Annahme einer globalen Semantik für Daten ist illusorisch. Daher sind IFD-Mappings unverzichtbar.

4 Automatische Verteilung

Gemäß des in 3.3 vorgestellten Modells des Kommunikationsgraphen, bedeutet das Erstellen eines Kommunikationsstruktur das Abbilden von Kommunikationsknoten auf IFBs. Die IFBs wiederum müssen ebenfalls auf Chips in der Systemarchitektur positioniert und realisiert werden.

In diesem Kapitel wird daher zuerst betrachtet, welche Auswirkungen die Position eines IFBs auf die Ressourcen der Systemarchitektur hat. Dann werden entsprechende Kostenfunktionen vorgestellt, um die Art und Größe der benötigten Ressourcen für die Realisierung einer Kommunikationsstruktur zu bestimmen. Dann folgt die Vorstellung der einzelnen Abschnitte eines Verfahrens, mit dem die benötigte Verteilung von IFBs auf Chips automatisiert werden kann.

4.1 Systemtopologien unter Verwendung von IFBs

Verschiedene Anzahlen von IFBs und verschiedene Positionierungen innerhalb einer Systemtopologie haben Auswirkungen auf die Kommunikationsverbindungen. Welche Auswirkungen das sind und welche Systemressourcen davon betroffen sind, wird in den beiden folgenden Abschnitten an einem Beispiel erläutert.

4.1.1 Verwendung eines einzelnen IFBs

Im IFS-Flow wird eine Menge von interagierenden Kommunikationskomponenten bestimmt. Wenn diese inkompatibel sind und über einen IFB kommunizieren sollen, wird die Position des IFB auf einem Chip in der Systemarchitektur vom Entwickler festgelegt. Von allen Kommunikationskomponenten müssen nun entsprechende Verbindungen vom Chip auf dem sie realisiert sind bis zum IFB gefunden werden (routing). Die Menge und Länge der benötigten Verbindungen hängt stark von der Position des IFBs innerhalb der Systemtopologie ab.

In Abbildung 4.1 ist eine Systemarchitektur zu sehen, in der auf dem Chip 3 ein IFB positioniert wurde. Die Pfeile, welche direkt Tasks verbinden, geben an welche Komponenten miteinander kommunizieren wollen. Durch die Systemarchitektur verbinden Pfeile IFBs und Tasks. Diese visualisieren die Anzahl und Länge der benötigten Verbindungen. Die Breite (Anzahl Bits) dieser Verbindungen ist nicht angegeben.

Wird der IFB auf einem Chip positioniert, auf dem sich Komponenten befinden, die über den IFB kommunizieren sollen, so müssen für diese Komponenten keine Verbindungen durch die Systemarchitektur bereitgestellt werden. Es müssen nur ausreichend lokale

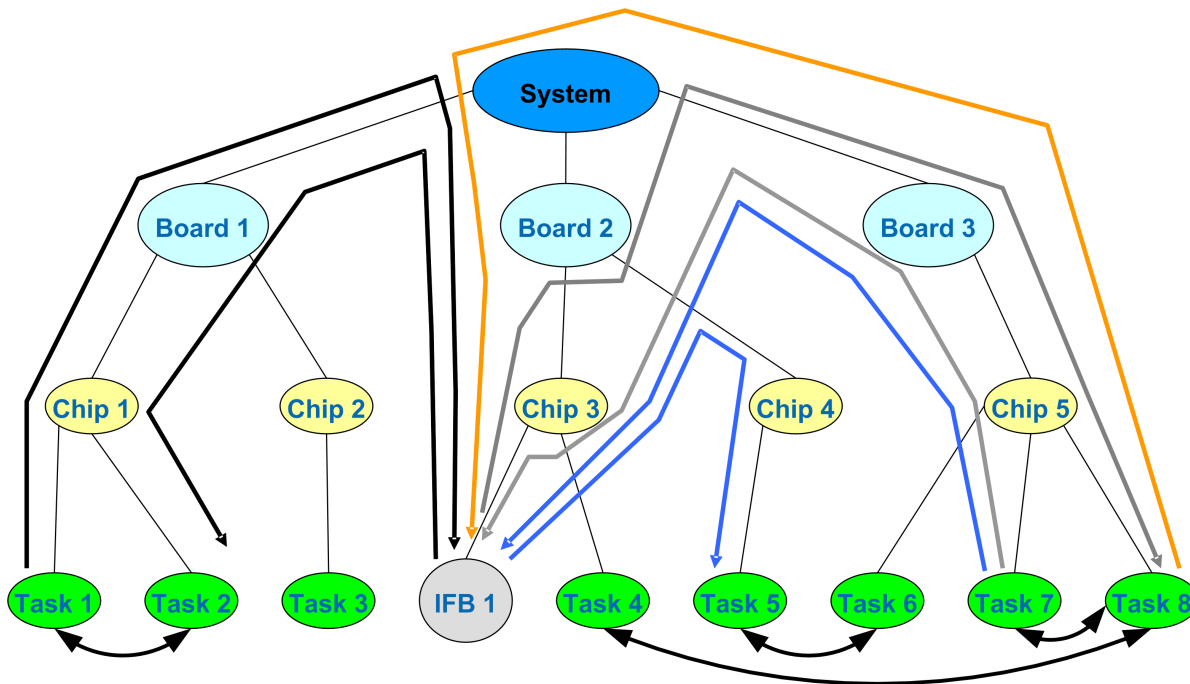


Abbildung 4.1: Systemtopologie unter Verwendung eines einzelnen IFBs

Verbindungsmöglichkeiten zwischen den Schnittstellen des IFBs und den Kommunikationskomponenten bestehen. Positioniert man hingegen für die gleiche Menge an Kommunikationskomponenten den IFB auf einem Chip im System auf dem sich keine Komponenten befinden, die über den IFB kommunizieren sollen, so ist die Anzahl und Länge der benötigten Kommunikationsverbindungen höher.

Die Anzahl der zur Verfügung stehenden Verbindungen zwischen den Komponenten der Systemarchitektur ist begrenzt. Kann nicht von allen Kommunikationskomponenten eine Verbindung zum IFB hergestellt werden, so kann die Kommunikationsstruktur nicht umgesetzt werden. Durch eine gezielte Positionierung eines IFBs innerhalb der Systemtopologie werden weniger Verbindungen durch das System benötigt, und es lassen sich komplexere Kommunikationsstrukturen realisieren.

4.1.2 Verwendung mehrerer IFBs

Die gezielte Positionierung eines einzelnen IFBs kann die Anzahl der benötigten Verbindungen zwar reduzieren, allerdings nur für eine Teilmenge der interagierenden Kommunikationskomponenten. Die Anzahl der benötigten Verbindungen kann weiter minimiert werden, indem mehrere IFBs verwendet werden. Über einen IFB kommuniziert dabei jeweils eine Teilmenge der Komponenten der Kommunikationsstruktur, welche alle untereinander interagieren. Ein Beispiel ist in Abbildung 4.2 zu sehen. Die interagierenden Komponenten sind die gleichen wie in Abbildung 4.1, aber die Anzahl der benötigten Verbindungen konnte stark verringert werden.

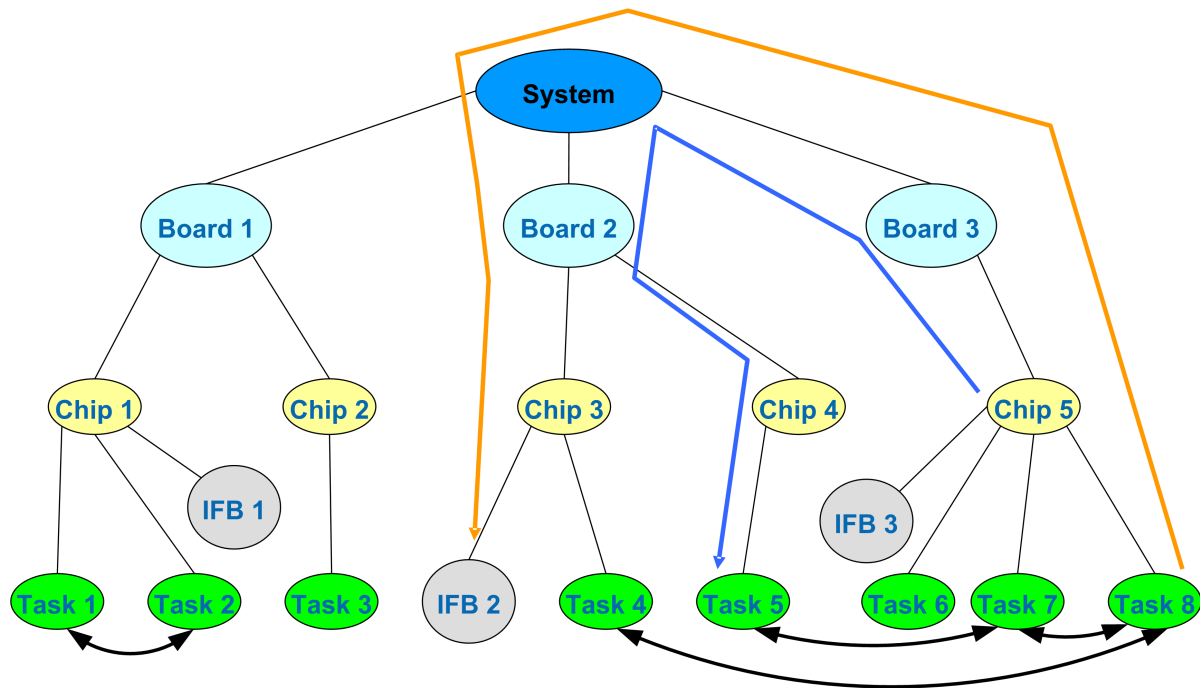


Abbildung 4.2: Systemtopologie unter Verwendung mehrerer IFBs

4.2 Automatische Verteilung von IFBs

Die automatische Verteilung von IFBs geht von einer festen Zuordnung von Tasks zu Chips und einer statischen Zusammenstellung von Kommunikationspartnern aus. Die zu erstellende Kommunikationsstruktur ist physikalischen Einschränkungen unterworfen, da die Systemarchitektur nur über eine begrenzte Anzahl von elektrischen Leitungen verfügt. Die verschiedenen Möglichkeiten der Zuordnung von IFD-Mappings zu IFBs, die Positionierung der IFBs auf einem der Chips der Systemarchitektur und die daraus resultierende Reservierung der notwendigen Verbindungen legen den Lösungsraum für ein Design der Kommunikationsstruktur fest.

Die Position eines IFBs innerhalb der Systemarchitektur bestimmt im hohen Maße die Anzahl der benötigten Verbindungen. Um eine komplexe Kommunikationsstruktur zu realisieren ist eine möglichst effiziente Ausnutzung der gegebenen Ressourcen und daher eine optimale Positionierung der IFBs in der Systemarchitektur erforderlich. Dieses Ziel wird erreicht, indem eine Kostenfunktion für Verbindungs- und Implementierungskosten eines IFBs aufgestellt wird. Anschließend wird ein Verfahren zur Minimierung dieser Kosten angegeben.

4.2.1 Kostenfunktionen

Die Gesamtkosten der Kommunikationsstrukturen in einer System-Architektur sind definiert als Summe aller Kommunikations- und Implementierungskosten. Diese Kosten

werden von nun an als $C_{SysArch}$ bezeichnet. Es gilt $C_{SysArch}$ zu minimieren, um das Erstellen einer Kommunikationsstruktur unter Berücksichtigung der gegebenen System-Architektur zu ermöglichen.

$$C_{SysArch} = C_{Tasks} + C_{StaticIO} + \sum_{i=1}^k C(IFB_i) \quad (4.1)$$

Die Zahl k ist die Anzahl der verwendeten IFBs und entspricht dem k aus der Gleichung (2.4.2). Für die Synthese der Tasks und die Allokation von direkten Verbindungen entstehen statische Kosten, welche aber für die Optimierung der Gesamtkosten nicht berücksichtigt werden müssen. Nur die Kosten, welche bei der Umsetzung von IFBs anfallen, sind variabel. Im folgenden wird zwischen zwei Arten von Kosten unterschieden:

1. Ein IFB benötigt Ressourcen in Form von konfigurierbaren Logikblöcken (CLB) oder Speicherplatz für seine Implementierung ($\Rightarrow$ *Implementierungskosten* C_{Impl}).
2. Zum Aufbau der Kommunikation zwischen dem IFB und den verbundenen Tasks werden Leitungen benötigt, um diese Verbindungen zu ermöglichen ($\Rightarrow$ *Verbindungskosten* $C_{Connect}$).

$$C(IFB_i) = C_{Impl}(IFB_i) + C_{Connect}(IFB_i) \quad (4.2)$$

Die Höhe der einzelnen Kosten ist von der Anzahl der IFD-Mappings, welche in einem IFB realisiert werden, abhängig. Für die weitere Erläuterung von C_{Impl} und $C_{Connect}$, gilt für einen IFB_i :

$$n = \#IFD\text{-Mapping zugewiesen zu } IFB_i \quad (4.3)$$

Implementierungskosten

Die *Implementierungskosten* C_{Impl} sind wie folgt definiert:

$$\begin{aligned} C_{Impl}(IFB_i) = & (1 + n \cdot IFB_{MappingKosten}) \cdot IFB_{Struktur} \\ & + \sum_{x=1}^{\#PH_{Modes}} C(PH_{M_x}) + \sum_{x=1}^{\#SH_{Modes}} C(SH_{M_x}) \end{aligned} \quad (4.4)$$

Der erste Summand der Gleichung beschreibt die Kosten, welche für die Implementierung der IFB Grundstruktur anfallen. Dazu gehören die CU, der Speicher für die Zwischenspeicherung der Datenpakete gemäß dem IFD-Mapping sowie Bussysteme und Steuerleitungen für interne Signale. Für jedes IFD-Mapping werden eine erweiterte Kontrollstruktur, weiterer Speicherplatz und zusätzliche Steuerleitungen benötigt.

Die Kosten erhöhen sich um den Faktor $IFB_{MappingKosten}$, welcher kleiner eins ist. Ist kein IFD-Mapping vorhanden ($n = 0$), so ist dieser Teil dennoch größer Null, da auch ohne IFD-Mappings Grundkosten für die CU anfallen. In diesem Fall kann die Implementierung des IFB jedoch entfallen. Der zweite Teil der Gleichung summiert die Kosten aller benötigten PH und SH Modes auf.

Verbindungskosten

Die *Verbindungskosten* $C_{Connect}$ sind wie folgt definiert:

$$C_{Connect}(IFB_i) = \sum_{m=1}^n \#conductions_m \cdot Länge_m \quad (4.5)$$

Die Anzahl der Kanten auf dem Pfad durch den Systemarchitekturgraphen ergibt die Länge einer Verbindung m ($Länge_m$). Für die Berechnung der Verbindungskosten könnten auch weitere Eigenschaften, welche in der IFD zur Verfügung stehen, in die Gleichung eingehen. So können bestimmte Konstellationen bevorzugt werden. Beispiele für diese Eigenschaften sind z.B. die Frequenz oder der Wert für den Jitter. Diese Eigenschaften wurden aber im Rahmen dieser Arbeit nicht weiter betrachtet.

Mit Hilfe der angegebenen Gleichungen können die Kosten $C(IFB)$ für einen IFB bestimmt werden.

4.2.2 Strategien für die Verteilung von IFBs

In heutigen Chips und FPGAs stehen eine riesige Menge an Transistoren und anderer Implementierungseinheiten zur Verfügung. Die Ressourcen der Kommunikation wurden im Laufe der Zeit immer mehr ein begrenzender Aspekt des SoC-Design (System-on-Chip).

Bei Entwicklungen auf FPGAs stehen innerhalb eines FPGAs nur eine sehr begrenzte Anzahl von langen Leitungen, welche über die gesamte Ausdehnung der FPGA Struktur laufen, zur Verfügung. Die I/O Pins für die off-chip Kommunikation eines FPGAs sind ebenfalls eine begrenzte Ressource. Besonders Kommunikationsverbindungen off-chip durch die System-Architektur sind daher extrem kostenintensiv und müssen somit minimiert werden.

Die Reduzierung der Verbindungskosten ist daher ein Hauptziel bei der Verteilung von IFBs. Reduzierte Verbindungskosten führen jedoch zu einer erhöhten Anzahl von IFBs, was wiederum in höheren Implementierungskosten resultiert. Abhängig von der gegebenen Systemarchitektur und dem aktuellen Designziel können verschiedene Strategien der Verteilung angemessen sein, wie in Abbildung 4.3 dargestellt wird.

1. Minimale Verbindungskosten

Für jede Kommunikation zwischen einer Menge von interagierenden Tasks wird ein IFB zur Verfügung gestellt und so auf einem der Chips auf dem sich die Tasks befinden positioniert, dass die Verbindungskosten minimal sind. Diese Strategie wird den IFB auf dem Chip platzieren, auf welchem sich die Task mit der höchsten benötigten Anzahl an Verbindungen befindet.

Der Nachteil dieser Strategie ist, dass jeder Menge kommunizierender Tasks ein IFB zur Verfügung gestellt wird und daher die Implementierungskosten sehr hoch sind.

2. Minimale Implementierungskosten

Die Anzahl der IFBs wird auf ein Minimum reduziert, und die Anzahl der IFD-Mappings pro IFB wird maximiert. Die Auslastung der übrig bleibenden IFBs wird dadurch erhöht und die Implementierungskosten auf ein Minimum reduziert. Der Nachteil dieser Strategie ist, dass sie zu sehr hohen Verbindungskosten führen kann.

3. Obere Schranke für die Anzahl an IFBs

Diese Strategie optimiert die Verbindungskosten unter der Berücksichtigung einer gegebenen oberen Schranke für die Anzahl der IFBs.

4. Obere Schranke für die Implementierungskosten eines IFB

Diese Strategie begrenzt die lokalen Implementierungskosten durch das Setzen einer oberen Schranke für die Implementierungskosten eines IFBs. Überschreitet ein IFB diese Schranke werden IFD-Mappings aus dem IFB entfernt und in einem anderen IFB realisiert, in dem noch ausreichend Ressourcen zur Verfügung stehen.

5. Einhalten einer Ausgeglichenheitsfunktion

Mit Hilfe einer Ausgeglichenheitsfunktion, welche von jedem IFB des Systems erfüllt sein muss, werden extreme Werte für Verbindungs- und Implementierungskosten vermieden. Ein wichtiges Ziel, welches mit dieser Strategie erreicht werden kann, ist eine Mindestauslastung aller IFBs. Dazu kann man eine Ausgeglichenheitsfunktion angeben, welche eine Mindestanzahl von IFD-Mappings pro IFB fordert. Sind in einem IFB nicht ausreichend viele IFD-Mappings realisiert, so werden die noch vorhandenen IFD-Mappings entfernt und in benachbarte IFBs verschoben. Durch diese Strategie wird z.B. das Verwenden eines IFB für nur ein einziges IFD-Mapping vermieden.

Jedes Kreuz in der Abbildung 4.3 repräsentiert eine mögliche pareto optimale Lösung für eine feste Anzahl an IFBs. Der Kreisbogen definiert die Linie auf welcher die Gesamtkosten c als Summe von Verbindungs- und Implementierungskosten entsprechend der Gleichung (4.1) gleich sind (Äquipotentiallinie). Der Absolutwert $|\vec{p}|$ definiert die Kosten einer Lösung, welche für den Punkt P des Lösungsraumes anfallen. Die Optimierungsstrategien können in die Kostenfunktion integriert werden. Dazu werden die Quantoren a und b in die Terme der Implementations- und Verbindungskosten der Gleichung (4.2) eingefügt. Die Gesamtkosten werden dadurch zu einer gewichteten Summe, die sich je nach Wahl der Quantoren an die Strategie 1-5 anpassen lässt.

$$C(IFB_i) = a \cdot C_{Impl}(IFB_i) + b \cdot C_{Connect}(IFB_i) \quad (4.6)$$

4.2.3 Clustering als Verfahren zur Reduktion der IFB-Anzahl

Das Problem der Verteilung einer Anzahl von IFBs über die Systemarchitektur und die Entscheidung welches IFD-Mapping in welchem IFB implementiert wird, kann abstrakt als Partitionierendes Clustering Problem aufgefasst werden.

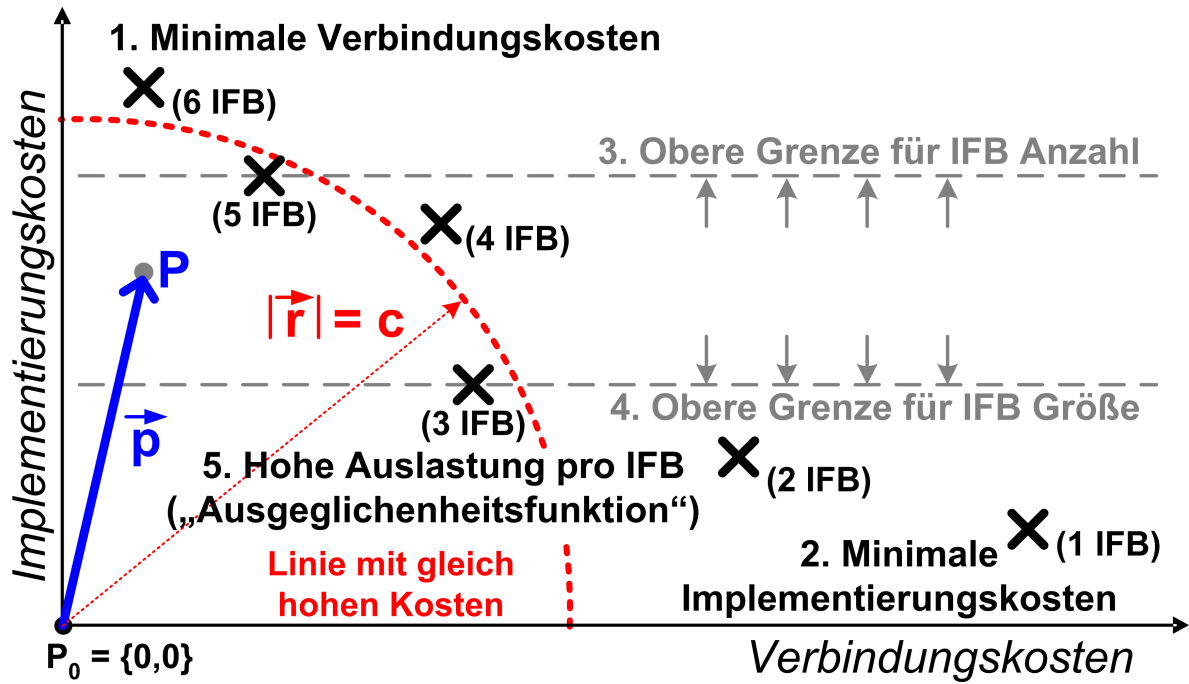


Abbildung 4.3: Strategien für die Verteilung von IFBs

Die Anzahl der möglichen Aufteilungen von n Elementen auf k Cluster wird durch die Gleichung (2.4.2) beschrieben. Alle möglichen Aufteilungen zu berechnen und zu bewerten ist meist nicht möglich, da die Anzahl der möglichen Aufteilungen sehr hoch sein kann. Das Berechnen aller Möglichkeiten wird noch zusätzlich dadurch erschwert, dass die Anzahl der Cluster k in dem hier betrachteten Ansatz Teil des Ergebnisses ist.

Es wird ein partitionierendes bottom-up Clustering verwendet, um dieses Problem zu umgehen. Der Algorithmus, welcher dieses Clustering realisiert, verwendet eine Heuristik, so dass nur eine kleine Anzahl aller möglichen Aufteilungen betrachtet wird. Das Ergebnis ist eine Anzahl disjunkter Cluster ohne eine Hierarchie. Jedes Cluster wird durch einen IFB repräsentiert. Die auf die Cluster verteilten Elemente sind die IFD-Mapping-Gleichungen, welche in den IFBs implementiert werden.

4.3 Ablauf der Verteilung von IFBs

Nachdem in den vorangehenden Abschnitten Aspekte wie Strategien, Ziele und Kosten für die automatische Verteilung von IFBs erläutert wurden, wird nun der Ablauf des Verfahrens selbst dargestellt.

4.3.1 Erstellen einer initialen Verteilung

Der heuristische Algorithmus beginnt mit dem Erstellen einer initialen Verteilung der IFBs mit einer entsprechenden Zuordnung der IFD-Mappings. Ein IFD-Mapping wird genommen und die Chips der Sender- und Empfängerseite der im IFD-Mapping aufgeführten Tasks bestimmt. Welche das sind, geht aus den Angaben des IFD-Mappings hervor. Ein IFB (= Cluster) wird auf dem Chip platziert, welcher die Empfänger Task eines IFD-Mappings enthält.

Der IFB wird nur auf dem Chip platziert, wenn die Verbindungs- und Implementationskosten die vorhandenen Ressourcen nicht überschreiten. Wenn das Platzieren auf der Empfängerseite nicht möglich ist, wird der IFB zur Senderseite migriert. Sollte das ebenfalls nicht möglich sein, wird der IFB nicht platziert und das IFD-Mapping, nach Verteilung aller anderen IFBs, im nächstgelegenen IFB implementiert.

Welcher IFB der nächstgelegene ist kann mit den Verbindungskosten, welche nur für diese eine IFD-Mapping entstehen, bestimmt werden. Für ein IFD-Mapping m sind die Verbindungskosten für die Realisierung im IFB i :

$$C_{Connect}(IFB_i) = \#conductions_m \cdot Länge_m \quad (4.7)$$

Der IFB, beim dem die geringsten Verbindungskosten entstehen, ist der nächstgelegene IFB.

Wenn bei der Initialisierung zwei IFBs auf dem selben Chip platziert werden sollten, werden sie zu einem einzigen IFB zusammengeführt, indem alle IFD-Mappings in einem IFB zusammengefasst werden. Dies wird aber nur vorgenommen, wenn die Schedulability Analyse für diesen zusammengefassten IFB erfolgreich ist. Die Schedulability Analyse ist Teil des IFS-Design-Flows.

Das so erzeugte initiale Cluster und die weiteren Optimierungsschritte führen dazu, dass ein Großteil des Lösungsraumes für die Verteilung von IFBs nicht betrachtet wird.

4.3.2 Optimierung

Nach Erstellen der initialen Verteilung wird eine Optimierung vorgenommen, um die Verbindungskosten zu minimieren. Ein IFD-Mapping welches nur Tasks verbindet, die auf dem selben Chip positioniert sind, wird als *lokales IFD-Mapping* bezeichnet.

Lokale IFD-Mappings erzeugen keine Verbindungskosten für den IFB, da die Länge der Verbindungen gleich Null ist. Das Migrieren eines lokalen IFD-Mappings zu einen anderen IFB auf einem anderen Chip resultiert immer in einem Anstieg der Verbindungskosten. Daher werden bei der Optimierung der initialen Verteilung von IFBs nur IFBs berücksichtigt, welche nur nicht lokale IFD-Mappings beinhalten.

Bei solchen IFBs wird versucht alle beinhalteten IFD-Mappings in andere IFBs zu migrieren. Diese Migration wird nur durchgeführt, wenn die Verbindungskosten dadurch nicht erhöht werden. Wenn auf diese Weise alle IFD-Mappings aus einem IFB entfernt werden konnten, wird der IFB aus dem System entfernt.

Nach dieser Optimierung erfüllt die so erreichte Verteilung die Zielsetzung der Strategie „*Minimale Verbindungskosten*“.

4.3.3 Auswahl der Strategie

Nach der Optimierung wird die Strategie für den weiteren Ablauf des Clustering-Algorithmus ausgewählt. Die Strategie beeinflusst den Clustering-Algorithmus in Bezug auf die Modifikation der Kostenfunktion wie in Abschnitt 4.2.2 beschrieben. Die gewählte Strategie hat auch Auswirkungen auf die verwendete Abbruchbedingung des Algorithmus.

4.3.4 Clustering Algorithmus

Der Cluster Algorithmus versucht die Implementierungskosten der initialen Verteilung so weit zu reduzieren, bis die Vorgaben der gewählten Strategie erfüllt sind oder keine weitere Reduktion mehr möglich ist.

Zum Reduzieren der Implementierungskosten werden alle IFD-Mappings von einem ausgewählten IFB migriert. Wenn alle IFD-Mappings migriert werden konnten wird der IFB entfernt. Eine verringerte Anzahl an IFBs führt zu verringerten Implementierungskosten. Da die Tasks fest an den Chip gebunden sind, werden durch die Migration der IFD-Mappings die Verbindungskosten erhöht. Um die Verbindungskosten so wenig wie möglich zu erhöhen und um überhaupt alle IFD-Mappings eines IFB entfernen zu können, müssen verschiedene Aspekte berücksichtigt werden:

- Die Migration von lokalen IFD-Mappings zu anderen IFBs führt meistens zu einer größeren Zunahme der Verbindungskosten als das Migrieren von nicht lokalen IFD-Mappings.
- Die Implementierungskosten können nur reduziert werden, wenn alle IFD-Mappings eines IFB migriert werden können und der IFB anschließend entfernt wird.

Da die Ressourcen zur Implementation der IFD-Mappings beschränkt sind, ist es nicht immer möglich alle IFD-Mappings eines ausgewählten IFB zu migrieren. Daher ist die Wahrscheinlichkeit einen IFB mit wenigen IFD-Mappings zu leeren größer als bei einem IFB mit vielen IFD-Mappings. Für die Entscheidung von welchem IFB die IFD-Mappings migriert werden sollen, wird das Entscheidungskriterium *interner* und *externer Grad* eines IFB definiert.

Der Wert des internen Grades ergibt sich aus der Anzahl aller Verbindungen die, aufgrund von lokalen IFD-Mappings eines IFBs zwischen Tasks und PH-Modes, benötigt werden. Der externe Grad repräsentiert die Anzahl der benötigten Verbindungen für alle nicht lokalen IFD-Mappings eines IFB.

Der Algorithmus startet mit dem IFB, welcher den niedrigsten internen Grad hat. Falls es mehrere IFBs mit dem niedrigsten internen Grad gibt, wird davon der IFB mit dem niedrigsten externen Grad ausgewählt. Die Migration der IFD-Mappings dieses so ausgewählten IFBs findet unter den gleichen Bedingungen statt wie in Abschnitt 4.3.1 und 4.3.2 beschrieben.

Wenn der IFB entfernt werden konnte wird die Abbruchbedingung geprüft und der nächste IFB über den Grad bestimmt. Wenn der IFB nicht entfernt werden konnte,

wird er markiert damit er nicht sofort nochmals ausgewählt werden kann. Dann wird der nächste IFB bestimmt.

4.4 Integration im IFS-Editor

Der in 3.1 Erwähnte IFS-Editor ermöglicht in weiten Bereichen bereits die technische Realisierung des IFS-Flow. Die Erstellung und manuelle Positionierung eines einzelnen IFBs ist mit Hilfe eines Assistenten im IFS-Editor möglich. In diesem Assistenten ist das in 4.3 erstellte Verfahren eingebunden (siehe Abbildung). Der Assistent wird auch die Auswahl der gewünschten Strategie zur Verteilung ermöglichen.

Das erstellte Verfahren bringt auch Vorteile, wenn man nur einen einzigen IFB erzeugen will. Man kann als Strategie für die Verteilung die *obere Schranke für die Anzahl an IFBs* wählen und dann diese Schranke auf eins setzen. So erhält man eine kostenminimale automatische Positionierung.

Die Menge der Kommunikationskomponenten und welche Komponenten miteinander kommunizieren wird im IFD-Mapping angegeben. IFD-Mappings sind nicht vollständig im IFS-Format spezifiziert und können noch nicht mit dem IFS-Editor erstellt oder verarbeitet werden (Stand: 08.03.2005). Daher war das erstellte Verfahren zum Zeitpunkt der Fertigstellung dieser Arbeit noch nicht vollständig funktional im IFS-Editor eingebunden.

Die Implementierung und Integration des IFD-Mappings in den IFS-Editor wird aber voraussichtlich im zweiten Quartal 2005 abgeschlossen, so dass die funktionale Integration der automatischen Verteilung kurze Zeit später abgeschlossen werden kann.

5 Dedicated Protocol

5.1 Ansätze für die Verwendung eines Dedicated Protocols

Wenn der Clustering Algorithmus stoppt, weil keine weitere Migration von IFD-Mappings mehr möglich ist, kann der Grund dafür das Fehlen freier Verbindungen in der Systemarchitektur sein. In diesem Fall könnte ein *Dedicated Protocol* verwendet werden, um die Verbindungskosten zwischen vorhandenen IFBs zu reduzieren. Dies wird durch das Zusammenfassen von zwei Protokollen zu einem Dedicated Protocol erreicht.

Ein solches Protokoll verwendet nur das Maximum der benötigten Verbindungen der beiden gegebenen Protokolle, anstelle der Summe. Wenn durch die Verwendung eines Dedicated Protocols die Anzahl der verwendeten Verbindungen reduziert werden konnte, kann der Clustering Algorithmus fortgesetzt werden.

5.2 Merged Protocol als Realisierung eines Dedicated Protocols

Für die Berechnung eines Merged Protocols aus zwei Protokollen müssen die folgenden Beschränkungen eingehalten werden:

- Die gemeinsame Verwendung von Verbindungen erlaubt nur die Ausführung eines Grundblocks aus einem Protokoll zu einem Zeitpunkt, während das andere Protokoll warten muss.
- Die Transitionen zwischen den Grundblöcken dürfen keine zeitlichen Abhängigkeiten beinhalten.
- Das Wechseln der Ausführung von einem Protokoll zu einem anderen ist nur am Ende eines Grundblocks möglich, damit die Verarbeitung von Datenpaketen nicht unterbrochen wird.
- Das Merged Protocol muss ein deterministisches Verhalten aufweisen.

Der hier vorgestellte Ansatz für Dedicated Protocols kombiniert zwei Protokolle zu einem neuen sogenannten *Merged Protocol*. Dazu werden die CFGs der beiden Protokolle zu einem neuen CFG kombiniert. Kombinieren zweier CFGs bedeutet das *Einfügen gültiger Transitionen* zwischen die Grundblöcke der beiden Protokolle P_A und P_B . Daher

ist es nicht möglich von einem inneren Zustand eines Grundblocks zwischen P_A und P_B zu wechseln.

Ein Merged Protocol kann nicht durch einfaches Erzeugen der Verbindung der beiden CFGs erstellt werden. Das Ergebnis wäre die Vereinigung der beiden Graphen mit einer neuen Kante von jedem Knoten des ersten Graphen zu jedem Knoten des zweiten Graphen. Mit Ausnahme von trivialen Fällen (nur ein Grundblock im CFG) würde dieser neue Graph ungültige Pfade enthalten. Die Pfade in einem Merged Protokoll P_{AB} müssen alle gültig sein.

Ein einzelner Pfad in einem Graphen eines Merged Protocols P_{AB} ist gültig, genau dann, wenn alle Abfolgen von Grundblöcken aus P_A und P_B einer möglichen Abfolge in P_A oder P_B entsprechen. Es darf also im Merged Protocol keine Abfolge von Zuständen auftreten, die in den ursprünglichen Protokollen nicht möglich war.

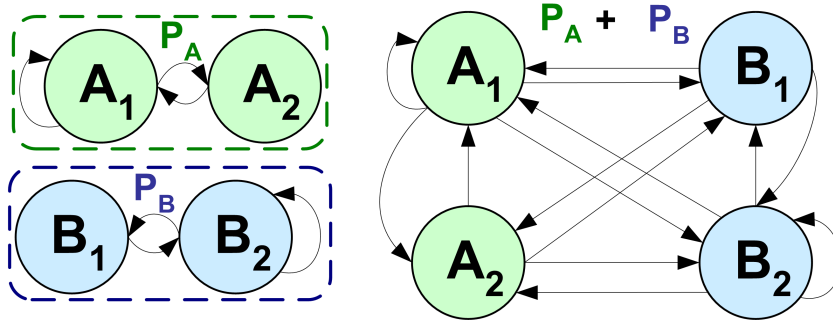


Abbildung 5.1: Verbindung von P_A und P_B mit ungültigen Pfaden

In Abbildung 5.1 ist ein Beispiel für die Verbindung der CFGs von Protokoll A und Protokoll B angegeben ($P_A + P_B$). Diese Verbindung enthält ungültige Pfade. Der Pfad A_2, B_2, A_2 würde zweimal den Grundblock A_2 besuchen, ohne dass zwischendurch der Grundblock A_1 vorkam. Diese Abfolge ist im Ursprungsprotokoll P_A nicht möglich und daher ungültig.

Um keine ungültigen Pfade in einem Merged Protocol zu erzeugen, gibt es zwei mögliche Ansätze:

1. Die Transitionsbedingung einer neu eingefügten Transition wird um eine zusätzliche Bedingung erweitert. Dazu wird ein zusätzliches Element, welches den letzten ausgeführten Grundblock für jedes Protokoll speichert, mit in der Transitionsbedingung berücksichtigt. Die erweiterte Transitionsbedingung wird dann zusätzlich zu den ursprünglichen Bedingungen beachtet. Eine Transition schaltet dann abhängig davon, ob der letzte für ein Protokoll ausgeführte Grundblock ein legaler Vorgänger zu dem Ziel-Grundblock der Transition ist. Der aus dieser Methode resultierende Graph ist, bis auf die geänderten Transitionsbedingungen, vergleichbar mit der Verbindung von zwei Graphen.
2. Transitionen, die zu einer ungültigen Abfolge von Grundblöcken führen, darf es zu keinem Zeitpunkt der Ausführung geben. Beim Einfügen einer neuen Transition

zu einem Grundblock aus Protokoll A müssen daher immer die Vorgänger aus dem gleichen Protokoll berücksichtigt werden. In dieser Form der Lösung werden Grundblöcke geklont, so dass die Information über die Abfolgen der Vorgänger im Pfad selbst kodiert wird. Der resultierende CFG enthält daher alle möglichen gültigen Abfolgen von Grundblöcken aus beiden Protokollen. Zusätzliche Elemente zum Speichern der Vorgänger oder Erweiterungen der Transitionsbedingungen sind nicht erforderlich. Das Ergebnis dieses Verfahrens ist ein Protokoll, welches von den vorhandenen Strukturen des IFS-Flows ausgeführt und interpretiert werden kann.

Das Erweitern der Transitionsbedingungen wäre verhältnismäßig aufwändig. Für jedes Merged Protocol wären zusätzliche Historyelemente zum Speichern des vergangenen Verlaufs der beinhalteten Protokolle notwendig. Diese Elementen müssten aufwändig beschrieben und letztendlich synthetisiert werden.

Das Kodieren der Informationen über den Verlauf im CFG selbst ist eleganter, da das berechnete Merged Protocol wie jedes andere Protokoll behandelt und mit vorhandenen Methoden synthetisiert werden kann. Daher wurde der zweiten Methode der Vorrang gegeben und ein Algorithmus zur automatischen Berechnung eines Merged Protocols entwickelt.

Ein Beispiel für ein berechnetes Merged Protocol ist in Abbildung 5.2 zu sehen. Oben in der Abbildung sind die CFGs der beiden Ausgangsprotokolle P_A und P_B zu sehen. Der Startgrundblock ist der Grundblock, welcher mit dem Startzustand des Protokolls beginnt. In P_A und P_B hat der Startgrundblock jeweils den Index 1. Beide Protokolle enthalten neben einfachen Transitionen zu nachfolgenden Grundblöcken jeweils eine Selbsttransition. Das aus diesen beiden Protokollen berechnete Merged Protocol ist im unteren Teil der Abbildung zu sehen. In jedem Zustand des Graphen P_{AB} ist jeweils eine Nummer in runden Klammern unter der Bezeichnung des Zustandes aufgeführt. Dies gibt die Reihenfolge an, in welcher die Zustände beim Erstellen des Graphen erzeugt wurden. Die Anzahl der Grundblöcke im Merged Protocol hat sich erhöht, um alle gültigen Abfolgen von Zuständen des CFG von P_A und P_B abbilden zu können.

5.3 Protokoll Parser

Der Algorithmus zum Erstellen eines Merged Protocol benötigt als Eingabe die CFGs der beiden Protokolle. Es werden daher die Grundblöcke der beiden Protokolle benötigt. Für diese Aufgabe wurde im Rahmen dieser Studienarbeit das Verfahren des *Protokoll Parsers* entwickelt.

Dieses Verfahren liefert zu einer gegebenen Protokoll-FSM eine Menge mit Grundblöcken. Zur Integration dieses Verfahrens in den IFS-Editor wurde eine Implementation in Java erstellt. Mit dieser Java Methode wurden verschiedene Aspekte für die Erstellung von Merged Protokollen evaluiert. Zudem kann der Protokoll Parser noch für eine Reihe anderer Bereiche im IFS-Editor verwendet werden, welche ebenfalls die Grundblöcke einer Protokoll-FSM benötigen.

Dazu gehören die Visualisierung von Protokollen im IFS-Editor und die Extraktion von Datenpaketen für das Erstellen eines IFD-Mappings. Zusätzlich werden beim Durchfüh-

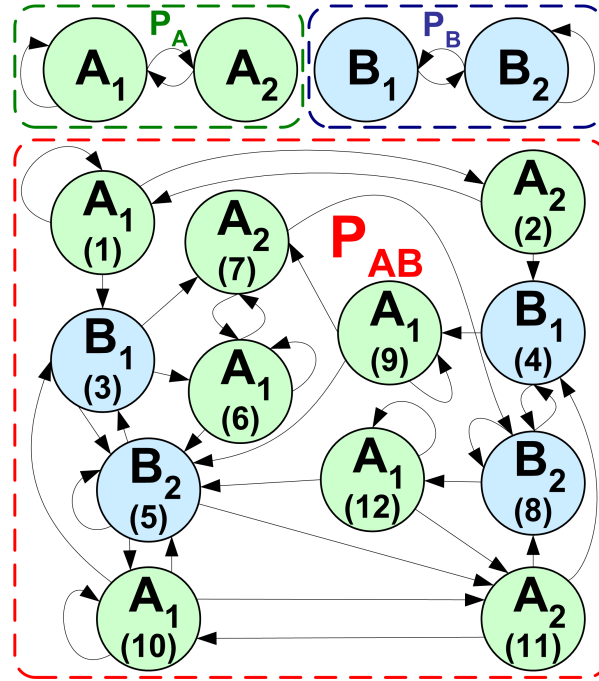


Abbildung 5.2: Merged Protocol

ren des Verfahren syntaktische Fehler eines eingegebenen Protokolls ermittelt, wenn diese sich in Form von unerreichbaren Zuständen in der Protokoll-FSM bemerkbar machen. Diese Art von syntaktischen Fehlern können durch *Überspezifikation* entstehen [PS91].

Das Prinzip des Ablaufs beim Protokoll Parser entspricht dem der *Tiefensuche*. In der folgenden Sequenz von Abbildungen ist der Ablauf des Algorithmus an einem Graphen einer Protokoll-FSM exemplarisch dargestellt. Die Ellipsen B1-B4 entsprechen dabei den gefundenen Grundblöcken.

Der Algorithmus erstellt zunächst einen Grundblock in dem nur der Startzustand des Protokolls liegt (Schritt 1). Solange es nur einen nachfolgenden Zustand gibt und dieser noch nicht in einem Grundblock liegt, werden die Nachfolger mit in den Grundblock aufgenommen. Sobald es mehrere Transitionen gibt, wird der aktuelle Grundblock als *abgeschlossen* markiert und für jeden Nachfolgezustand ein neuer Grundblock erstellt (Schritt 2).

Wenn der Nachfolgezustand bereits in einem Grundblock liegt (siehe S4 in Schritt 4), dann wird der aktuell bearbeitet Grundblock ebenfalls als *abgeschlossen* markiert und der Grundblock des Nachfolgezustands aufgeteilt (Bild 5). Dies wird solange durchgeführt, bis alle vorhandenen Grundblöcke als *abgeschlossen* markiert wurden.

Sind alle Grundblöcke als *abgeschlossen* markiert und es gibt noch Zustände die zu keinem Grundblock gehören, so handelt es sich um nicht erreichbare Zustände (*tote Zustände*). Diese Zuständen können durch Fehler in der eingegebenen Spezifikation des Protokolls entstehen.

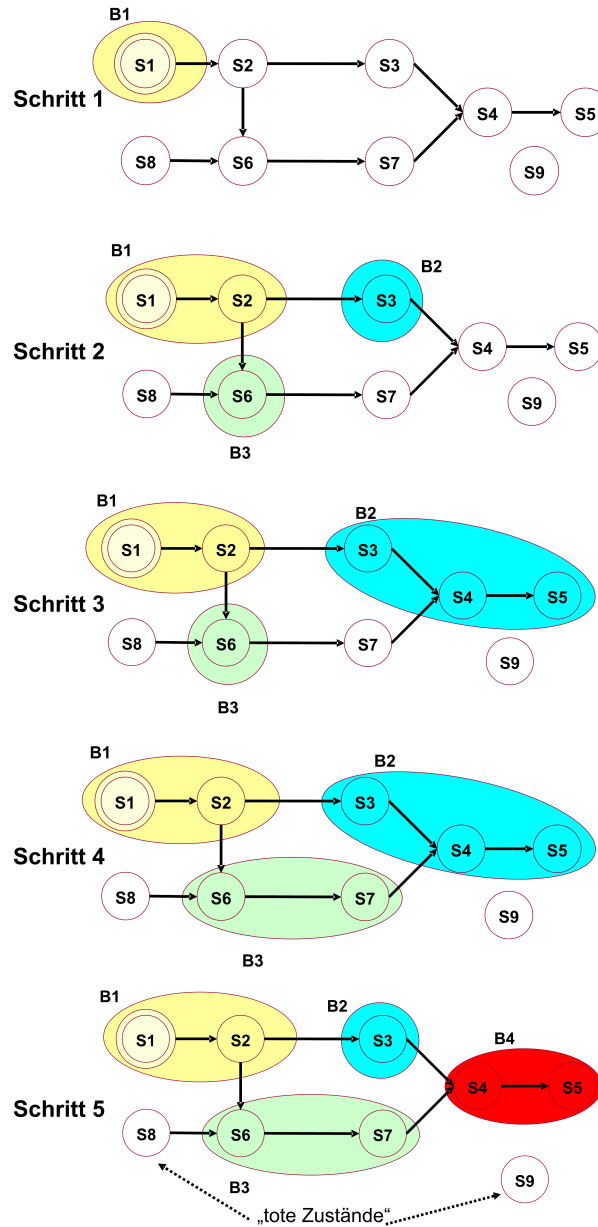


Abbildung 5.3: Protokoll Parser, Ablauf

5.4 Berechnung eines Merged Protocols

Die Berechnung eines Merged Protocols kann automatisch durch einen iterativen Algorithmus erfolgen. Zuerst werden dafür die CFGs von zwei übergebenen Protokollen erzeugt. Dies geschieht mit Hilfe des Protocol Parsers. Dann wird eine Pfad Komposition der beiden CFGs vorgenommen. Das Ergebnis ist der CFG eines Merged Protocols.

Der folgende Algorithmus arbeitet auf Listen von Grundblöcken. Ein Grundblock kann zwei verschiedene Zustände haben: *unbehandelt* oder *abgeschlossen*. Anfangs ist jeder neu

erzeugte Grundblock im Zustand *unbehandelt*. Wenn alle Transitionen zu allen nachfolgenden Grundblöcken generiert wurden und alle fehlenden nachfolgenden Grundblöcke des aktuell bearbeiteten Grundblocks erstellt sind, dann wird der aktuelle Grundblock (BB_{curr}) als *abgeschlossen* markiert.

Die Liste der Nachfolger des aktuellen Grundblocks ergibt sich aus den Nachfolgern welche der Grundblock im eigenen Protokoll hat. Hinter diese kommen die Nachfolgerlisten seiner Vorgänger, wobei der aktuelle Grundblock nicht selbst aus diesen Nachfolgerlisten übernommen wird, da sonst jeder Grundblock eine Selbsttransition hätte. Am Anfang des Algorithmus bekommt der erste Grundblock den Startgrundblock des anderen Protokolls mit in die Nachfolgerliste.

Pseudo Code für die Generierung eines Merged Protocol

```

01  $BB_{curr}$  = erster BB des ersten Protokoll
02 WHILE( $BB_{curr} \neq \text{null}$ ) DO
03   FOR( $i$  in  $\text{succ}(BB_{curr})$ ){
04     IF ( $\text{succ}(BB_{curr})[i]$  passt zu einem  $BB_{abgeschlossen}$ ) {
05       erzeuge Transition zu  $BB_{passend}$ 
06       aktualisiere( $\text{succ}(BB_{curr})$ ,  $\text{pred}(BB_{passend})$ ) }
07     ELSE IF( $\text{succ}(BB_{curr})$  passt zu einem  $BB_{unbehandelt}$ ){
08       erzeuge Transition zu  $BB_{passend}$ 
09       aktualisiere ( $\text{succ}(BB_{curr})$ ,  $\text{pred}(BB_{passend})$ ) }
10     ELSE {
11       erzeuge neuen  $BB_{neu}$ 
12        $\text{unbehandelt BB List.add}(BB_{neu})$ 
13       aktualisiere( $\text{succ}(BB_{curr})$ ,  $\text{pred}(BB_{neu})$ ) }
14   } // END FOR
15   markiere  $BB_{curr}$  als abgeschlossen
16    $BB_{curr} = \text{unbehandelt BB List.getNext}()$ 
17 } // END WHILE

```

Der Algorithmus verwendet die folgenden Definitionen und Funktionen:

- Die beiden übergebenen Protokolle werden mit P_1 und P_2 bezeichnet
- $\text{succ}(BB_x)$ liefert eine Liste mit allen nachfolgenden Grundblöcken von x
- $\text{pred}(BB_x)$ liefert eine Liste mit allen vorangehenden Grundblöcken von x
- $BB\ a \in P_1$ ist passend zu $BB_{abgeschlossen}\ b$ wenn $\text{succ}(b) = \text{succ}(a) / b$
- $BB\ a \in P_1$ ist passend zu $BB_{unbehandelt}\ b$ wenn gilt:
 - $\text{succ}(\forall BB: (\text{pred}(b) \in P_1)) \equiv \text{succ}(a)$

$$- \forall (\text{pred}(b) \in P_2) \in \text{pred}(a)$$

In der Tabelle 5.1 ist der Ablauf des Algorithmus und der Aufbau der verschiedenen Listen beim Erstellen des Merged Protocols aus P_A und P_B zu sehen. Es handelt sich um die gleichen Protokolle P_A und P_B wie in Abbildung 5.2. Die Tabelle zeigt den Zustand der einzelnen Listen jeweils nachdem der aktuelle Grundblock abgearbeitet und als abgeschlossen markiert wurde.

Nr.	abgeschlossen	unbehandelt	aktueller BB	Nachfolger
1	1	2,3	$A_1(1)$	A_1, A_2, B_1
2	1,2	3,4	$A_2(2)$	A_1, B_1
3	1,2,3	4,5,6,7	$B_1(3)$	B_2, A_1, A_2
4	1,2,3,4	5,6,7,8,9	$B_1(4)$	B_2, A_1
5	1,2,3,4,5	6,7,8,9,10,11	$B_2(5)$	B_1, B_2, A_1, A_2
6	1,2,3,4,5,6	7,8,9,10,11	$A_1(6)$	A_1, A_2, B_2
7	1,2,3,4,5,6,7	8,9,10,11	$A_2(7)$	A_1, B_2
8	1,2,3,4,5,6,7,8	9,10,11,12	$B_2(8)$	B_1, B_2, A_1
9	1,2,3,4,5,6,7,8,9	10,11,12	$A_1(9)$	A_1, A_2, B_2
10	1,2,3,4,5,6,7,8,9,10	11,12	$A_1(10)$	A_1, A_2, B_1, B_2
11	1,2,3,4,5,6,7,8,9,10,11	12	$A_2(11)$	A_1, B_1, B_2
12	1,2,3,4,5,6,7,8,9,10,11,12	-	$A_1(12)$	A_1, A_2, B_1, B_2

Tabelle 5.1: Merge Algorithmus

Die ersten drei Schritte des Merged Algorithmus sind in 5.4 zu sehen. Im ersten Schritt wird der Startgrundblock des Protokoll A in den neuen Merged Protokoll Graphen eingefügt. Er erhält die Nummer 1 (aktueller BB: $A_1(1)$). Die Nachfolgerliste ergibt sich aus den Nachfolgern von A_1 (A_1, A_2) und dem Startgrundblock aus Protokoll B (B_1).

Als nächstes werden die Transitionen zu den Grundblöcken der Nachfolgerliste eingefügt. Dabei wird als erstes versucht einen passenden *abgeschlossenen* Grundblock zu finden. Existiert solch ein passender Grundblock nicht, wird in der Liste der *unbehandelten* Grundblöcke weitergesucht. Für den Nachfolger A_1 des aktuellen Grundblocks wird nun ein passender Grundblock gefunden. In diesem Fall $A_1(1)$ selbst.

Für die restlichen Nachfolger A_2 und B_1 wird kein vorhandener Grundblock gefunden, daher werden entsprechend neue Grundblöcke ($A_2(2)$ und $B_1(3)$) erstellt. Da nun Transitionen zu allen Grundblöcken der Nachfolgerliste erstellt wurden, wird $A_1(1)$ als *abgeschlossen* markiert. Die Bearbeitung geht mit dem nächsten *unbehandelten* Grundblock ($A_2(2)$) weiter.

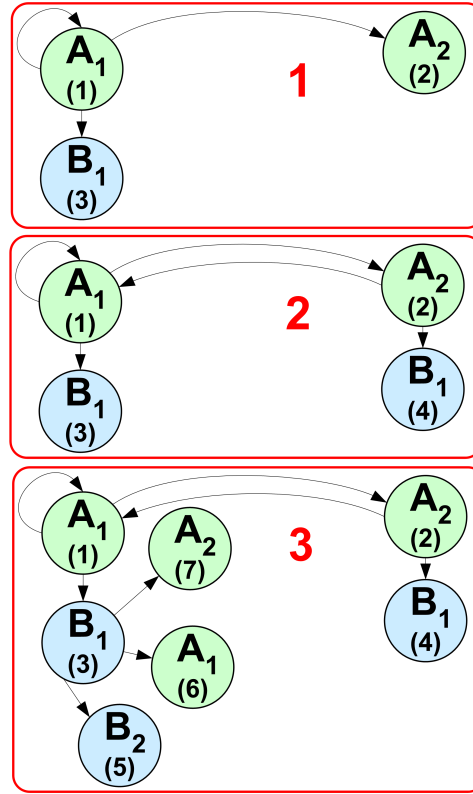


Abbildung 5.4: Merged Protocol 1-3

5.5 Auswirkung auf den IFB

Abbildung 5.5 zeigt zwei IFBs zwischen denen via eines Merged Protocols kommuniziert wird. In dem angegebenen Beispiel interagieren $T_1 \leftrightarrow T_3$ und $T_2 \leftrightarrow T_4$ miteinander. Alle Tasks verwenden inkompatible Protokolle. Die Konvertierung der Protokolle wird für $T_1 \leftrightarrow T_3$ und $T_2 \leftrightarrow T_4$ auf der für die Verbindungskosten optimalen Seite vorgenommen. Das bedeutet nur die kosten-minimalen Protokolle $P_A = \min(P_1, P_3)$ und $P_B = \min(P_2, P_4)$ müssen als Verbindungen durch die Systemarchitektur geführt werden.

Aus den so bestimmten Protokollen P_A und P_B wird das Merged Protocol P_{AB} generiert. Es existiert nur zwischen den beiden IFBs und ist somit „transparent“ für die Tasks. Die Anzahl der benötigten Verbindungen für P_{AB} reduziert sich auf das Maximum $\max(\#Verbindungen(P_A), \#Verbindungen(P_B))$.

Dabei ist zu beachten, dass es sich um eine Kommunikation zwischen zwei IFBs handelt und die Daten nur einmal gemäß dem Mapping umgewandelt werden müssen. Auf der Seite wo das Mapping nicht durchgeführt werden muss, werden SH-Modus generiert, welche die Daten einfach unverändert weiterreichen (*triviale SH-Modus*).

Die Anzahl der benötigten Verbindungen (Verbindungskosten) wird reduziert, es werden aber mehr PH-Modus benötigt. Daher kommt es zu einem leichten Anstieg der Im-

plementierungskosten. Da die Verbindungen in der Systemarchitektur die begrenzte Ressource sind, ist das nicht kritisch.

Ein Merged Protocol aus mehr als zwei Protokollen kann durch iteratives Anwenden des Algorithmus auf jeweils zwei Protokolle erstellt werden. Dabei reduziert sich die Anzahl der benötigten Verbindungen für $P_{A..X}$ auf $\max(\#Verbindungen(P_A), \dots, \#Verbindungen(P_X))$.

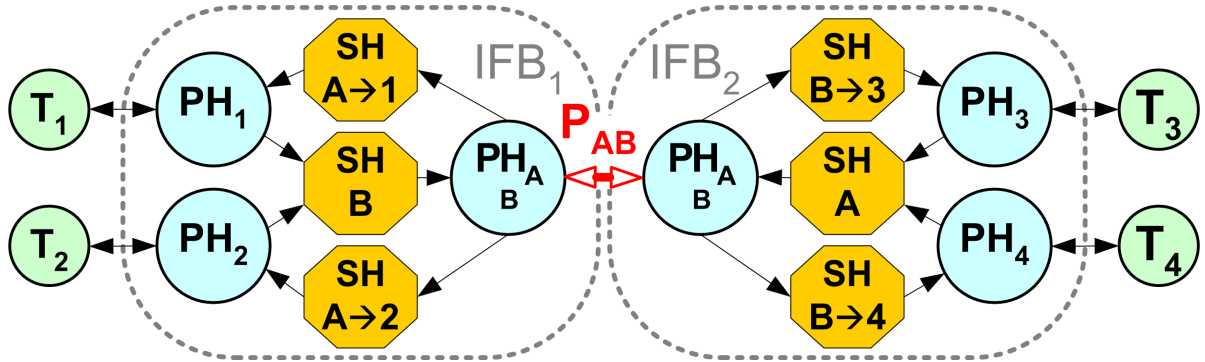


Abbildung 5.5: IFBs mit Merged Protocol

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung der Arbeit

Das Ziel dieser Arbeit war die Erstellung von Verfahren zur Unterstützung der Synthese von Kommunikationsstrukturen in verteilten eingebetteten Systemen mit Hilfe des IFS-Flows.

Dazu wurde zunächst das Modell des Kommunikationsgraphen basierend auf dem Spezifikationsgraph Modell vorgestellt. Für die Erstellung einer Kommunikationsstruktur gemäß dem Modell des Kommunikationsgraphen, ergaben sich die Anforderungen für eine automatische Verteilung und Abbildung der Kommunikationsknoten (realisiert durch IFBs) auf die Systemarchitektur.

Im weiteren Verlauf wurden die Auswirkungen der Positionierung von IFBs an unterschiedlichen Stellen der Systemarchitektur erläutert. Diese Auswirkungen resultierten in der Erstellung der verwendeten Kostenfunktionen. Für die automatisierte Verteilung wurde dann ein Clusteringverfahren erstellt, dass die automatische Verteilung von IFBs und die Zuordnung von IFD-Mappings ermöglicht. Basierend darauf wurden fünf verschiedene Verteilungsstrategien und ihre Auswirkungen auf die Kosten dargelegt.

Um Kommunikations-Engpässe zu überwinden, welche die automatische Verteilung einschränken, wurde das Prinzip des Merged Protocols vorgestellt. Dazu wurde der Algorithmus definiert mit dem ein Merged Protocol automatisch berechnet werden kann. Die erstellten Verfahren verbessern den IFS-Flow aufgrund der vereinfachten und effizienten Synthese von Kommunikationsstrukturen. Der Integration in das Konzept des IFS-Design-Flow wurde vorgestellt.

6.2 Ausblick auf weitere Modellierung

Die Präsentation der angegebenen Verfahren hat gezeigt, dass der IFS-Flow hinsichtlich der Synthese von Kommunikationsstrukturen erweitert werden konnte.

Für die Umsetzung des IFS-Flows wird der IFS-Editor verwendet. Mit ihm können die einzelnen Designschritte und die Synthese vorgenommen werden. Teile der in dieser Arbeit erstellten Verfahren sind bereits im IFS-Editor umgesetzt. Der im Rahmen dieser Arbeit entstandene Protokoll Parser wurde bereits vollständig. Er wird auch von anderen Bereichen des IFS-Editors verwendet. Im IFS-Editor können Protokolle mit Hilfe des Protokoll Parsers visualisiert werden. Ein weiterer wichtiger Bereich, für den der Protokoll Parser zukünftig benötigt wird, ist die Erstellung der IFD-Mapping Equations.

Die Verfahren zur automatisierten Verteilung und Erstellung von Merged Protokollen sind bereits an entsprechenden Stellen im IFS-Editor positioniert, aber noch nicht

vollständig funktional, da andere benötigte Komponenten noch nicht zur Verfügung stehen. Insbesondere fehlen IFD-Mapping equations. Ohne diese kann nicht ermittelt werden, welche Komponenten überhaupt miteinander kommunizieren wollen. Die Realisierung von IFD-Mapping equations wird aber in Kürze zur Verfügung stehen.

Nachdem IFD-Mappings mit dem IFS-Editor erstellt werden können, wird die zukünftige Arbeit im Bereich der Synthese von Kommunikationsstrukturen sich mit der Erstellung eines Beispiel Szenarios beschäftigen, um reale Werte für die vorgestellten Kostenfunktionen zu erhalten. Damit lassen sich dann die Größenordnungen für den Ressourcenbedarf der einzelnen Komponenten (wie z.B. PH-Modes oder Control Unit) ermitteln.

6.3 Ausblick für mögliche Erweiterungen

Nach der Ermittlung von Werten für die Kostenfunktion sind noch weitere Forschungsarbeiten im Rahmen von Studien- und Diplomarbeiten denkbar. Insbesondere sind die folgenden Themen geeignet, die sich im Verlauf der Arbeit ergeben haben, deren vollständige Ausarbeitung aber aus zeitlichen oder thematischen Gründen nicht möglich war:

- Optimierung der angegebenen Strategien für die automatische Verteilung von IFBs mit einer automatisierten Abänderung der Strategie, falls sich das Ziel der gewählten Strategie nicht erreichen lässt.
- Erweiterung des Verfahrens zur automatisierten Verteilung von IFBs, so dass die positionierten IFBs geforderte Kriterien für harte Realzeit einhalten können.
- Visualisierung der Verteilung bzw. des Clusterings von IFBs. Dabei könnte dem Benutzer die Möglichkeit zur Interaktion ermöglicht werden, um das Ergebnis aktiv zu beeinflussen.
- Erstellung eines Merged Protocols unter Verwendung der Erweiterung der Transitionsbedingungen. Diese würde das Synthetisieren von Elementen erfordern die den Verlauf (History) von Protokollen speichern können. Dieses Verfahren könnte Vorteile beim Erstellen eines Merged Protocols aus mehr als zwei Protokollen bieten, da sich bei diesem Verfahren die entstehende Anzahl der Zustände im wesentlichen auf die Summe der Zustände der einzelnen Protokolle beschränkt.

Literaturverzeichnis

- [AKJ88] Richard C. Dubes Anil K. Jain. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.
- [GNL67] W. T. Williams G. N. Lance. Mixed-data classificatory programs i - agglomerative systems. *Australian Computer Journal*, 1(1):15–20, January 1967.
- [Har69] F. Harary. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.
- [Ihm02a] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. A Consistent Design Methodology for Configurable HW/SW-Interfaces in Embedded Systems. In *Proc. of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems*, Montreal, Canada, Aug. 2002.
- [Ihm02b] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. A Design Methodology for Application-specific Real-Time Interfaces. In *Proceedings of 2002 IEEE International Conference on Computer Design (ICCD): VLSI in Computers & Processors*, IEEE International Conference on Computer Design, Freiburg, Germany, Sept. 2002.
- [Ihm03a] Ihmor, Stefan and Bastos Jr., Nilson and Klein, Rafael Cardoso and Visarius, Markus and Hardt, Wolfram. Rapid Prototyping of Realtime Communication - A Case Study: Interacting Robots. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, June 2003.
- [Ihm03b] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. Modeling of Configurable HW/SW-Interfaces. pages 51 – 60, 2003.
- [Ihm04] Ihmor, Stefan and Hardt, Wolfram. Runtime Reconfigurable Interfaces - The RTR-IFB Approach. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW'04)*. IEEE Computer Society, 26 - 27 April 2004.
- [KHK96] Friedrich Schwarz Karl-Heinz Kiyek. *Mathematik für Informatiker*, pages 32–33. B. G. Teubener Stuttgart, 1996.
- [PRSV98] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of Design Automation Conference*, pages 8–13, 1998.

- [PS91] Robert L. Probert and Kassam Saleh. Synthesis of communication protocols: Survey and assessment. *IEEE transactions on computers*, 40(4):468–469, April 1991.
- [Tei97] J. Teich. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer-Lehrbuch, Heidelberg, New York, Tokio, 1997.
- [ZRL96] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 103–114, Montreal, Canada, Juni 1996.

VHDL Codegenerierung für rekonfigurierbare Schnittstellen in eingebetteten Systemen

Gilles Bertrand Gnokam Defo

Die automatische Generierung von ausführbarem Zielcode ist ein wichtiger Schritt im Entwicklungsprozess eingebetteter Systeme. Eine automatisierte Codegenerierung beschleunigt zum einen diesen fehleranfälligen Entwurfsschritt und schafft das Potential die Wartbarkeit sowie die Ausnutzung plattformabhängiger Eigenschaften zu verbessern.

Die nachfolgende Arbeit beschreibt die Generierung von VHDL-Code für den Interface Block. Der Codegenerator ist derart ausgelegt, dass der aus dem IFS-Format (XML) erzeugte VHDL Code nahtlos in den Syntheseprozess für die Zielplattform XILINX-FPGA einbettet werden kann. Dabei wird insbesondere der Aspekt der Rekonfigurierbarkeit des Interface Blocks berücksichtigt.

Als Einleitung werden unterschiedliche Alternativen der Codegenerierung diskutiert. Für die Codegenerierung wird die Technik des "Frame Processing" ausgewählt, da sie für eine komponentenorientierte Sprache wie VHDL besonders geeignet ist. Die vorliegende Arbeit liefert einen Überblick über den gesamten Syntheseablauf und beschreibt die Integration der Codegenerierung in den Prozess der Schnittstellensynthese.

Inhaltsverzeichnis

1. Einführung	207
1.1. Motivation	207
1.2. Aufgabenstellung	208
1.3. Gliederung	208
2. Grundlagen	209
2.1. Pattern für die Codegenerierung	209
2.1.1. Einsatzgebiete der Codegenerierung	209
2.1.2. Pattern	209
2.2. Interface Synthesis Design-Flow (IFS-Flow)	213
2.2.1. System-Architektur	214
2.2.2. Target Platform Description (TPD)	214
2.2.3. Interface Description (IFD)	214
2.2.4. Das IFD-Mapping	215
2.2.5. IFS-Flow	215
2.3. Der Interface Block (IFB)	217
2.3.1. IFB-Makrostruktur	217
3. IFB-Synthese	219
3.1. IFB-Rekonfiguration	219
3.1.1. Rekonfiguration der Handler	220
3.1.2. Steuerung der Rekonfiguration durch die Control Unit	221
3.2. Erzeugung des Inputs für den Codegenerator	223
4. Codegenerierung	225
4.1. Analyse des IFB-Templates	225
4.1.1. Protocol Handler	225
4.1.2. Sequence Handler	226
4.1.3. Control Unit	228
4.1.4. Ergebnis der Analyse des IFB-Templates	228
4.2. Auswertung der Interface Descriptions (IFD)	229
4.3. Aufbau des Codegenerators	230
4.3.1. VHDL Gen	230
4.3.2. VHDL Factory	231
4.4. Erzeugung der VHDL-Pattern	231
4.5. Generierung einer Entity	231

4.6.	Generierung einer Architecture	233
4.6.1.	Generierung der Verhaltensbeschreibung	233
4.6.2.	Generierung der Strukturbeschreibung	237
4.7.	Synthese der Register-File	238
4.7.1.	Frames und Packages	238
4.7.2.	Register-Files	239
5.	Implementierung des Codegenerators	241
5.1.	Package-Diagramm des Codegenerators	241
5.2.	Klassendiagramme	242
5.2.1.	vhdlGen	242
5.2.2.	vhdlFactory	242
5.3.	Verwendete Datenstrukturen	243
5.3.1.	CodeList, CodeFragment, Tag	243
5.4.	Automatisierung: CodegenerationWizard	244
6.	Zusammenfassung und Ausblick	247
6.1.	Zusammenfassung	247
6.2.	Ausblick	247
A.	Anhang	249
A.1.	SynthesisWizard	249
	Literaturverzeichnis	253

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 Softwaretechnik eingesetzt. Ein Codegenerator im Compilerbau ist ein Teil eines Compilers, der während der Synthesephase eines Kompilervorgangs 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 eingesetzten 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 ersten 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 Kompilervorgangs 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 Kompilervorgangs 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 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

2. Grundlagen

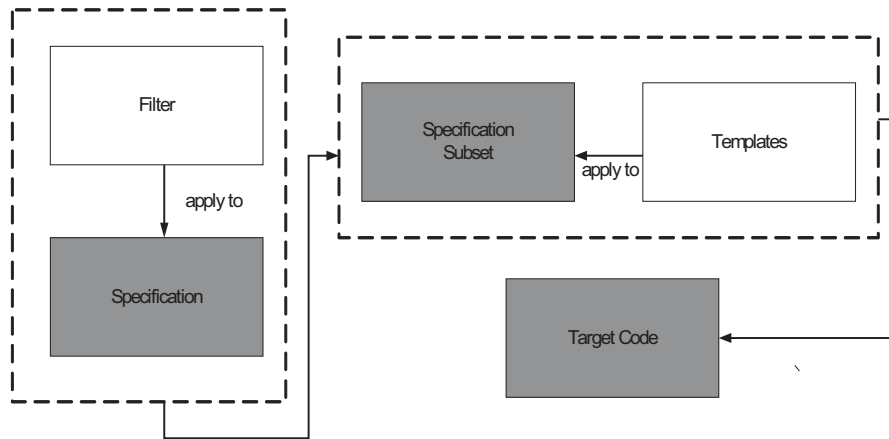


Abbildung 2.1.: Template + Filtering

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].

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 + Fil-

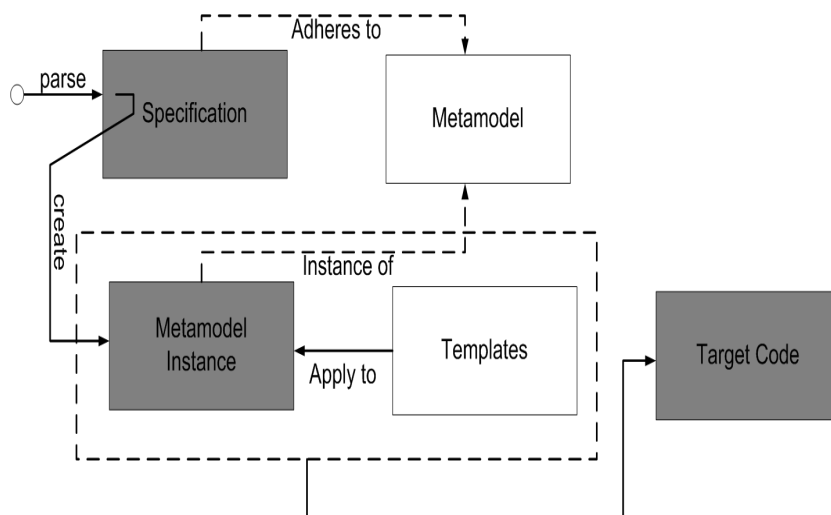


Abbildung 2.2.: Template + Metamodel

tering zur Verfügung.

Abbildung 2.2 zeigt, wie die Codegenerierung in zwei Schritten durchgeführt wird. Zuerst werden Metamodelle aus der Spezifikation instanziiert 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 in 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 instanziiert, parametrisiert und fügt die Instanzen aller Frames zusammen. Letztendlich evaluiert er die zusammengebaute Frame-Hierarchie, um den gewünschten Code (Target Code) zu generieren [Mar02].

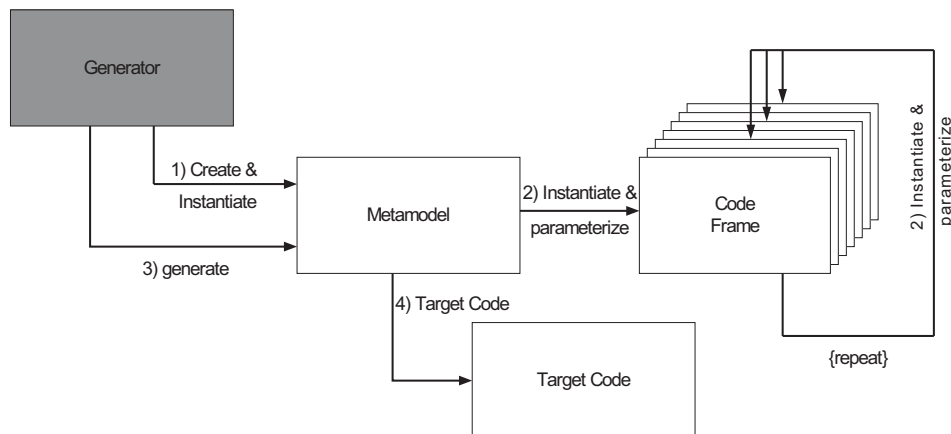


Abbildung 2.3.: Frame Processing

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 in Abbildung 2.4 dargestellt.

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

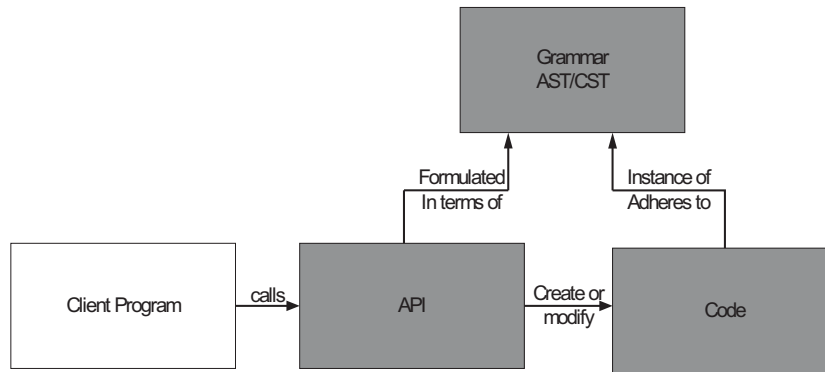


Abbildung 2.4.: API Based Generators

ersetzt, bevor diese ausgewertet werden. Der generierte Code wird so schrittweise zusammengesetzt und an einen spezifischen Kontext angepasst [Mar02].

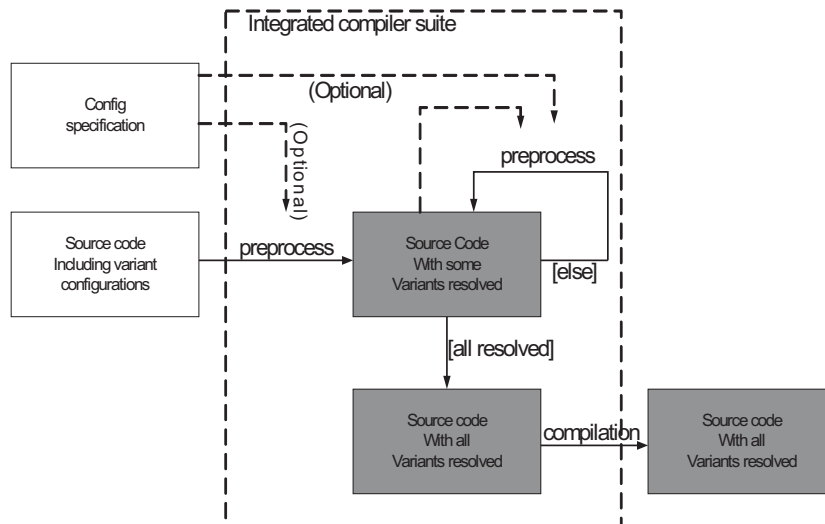


Abbildung 2.5.: Inline Code Generation

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 parsen, um die Erweiterung des Codes durchzuführen [Mar02]. Diese Technik wird in 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

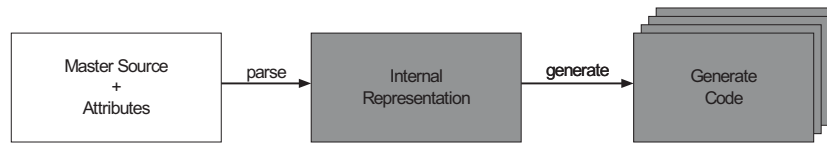


Abbildung 2.6.: Code Attributes

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].

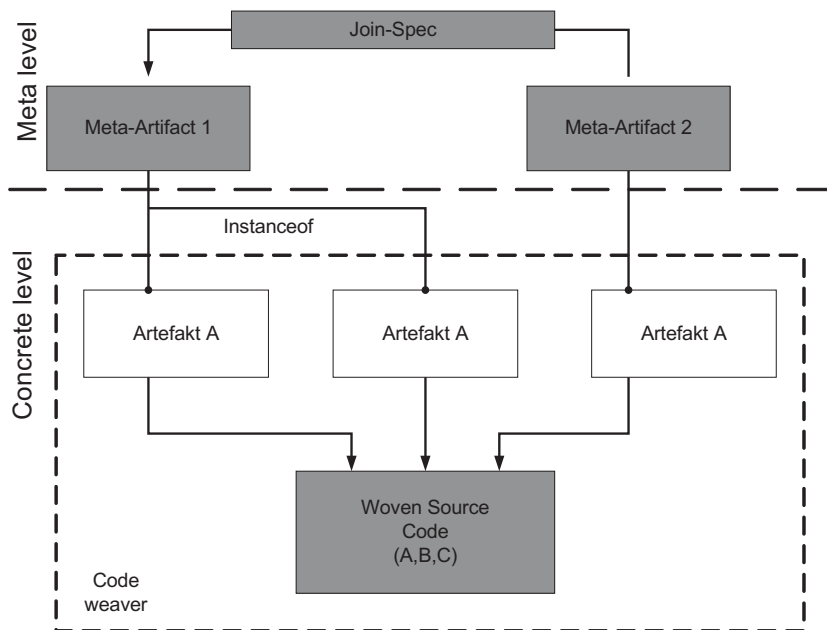


Abbildung 2.7.: Code Weaving

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 in 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.

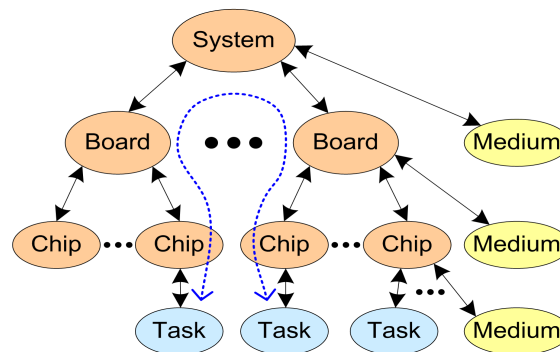


Abbildung 2.8.: Die System-Architektur

2.2.2. Target Platform Description (TPD)

Die Target Platform Description (TPD) beschreibt die Eigenschaften der Zielplattform, auf der der generierte 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:

$$MapEqn : data_{Out} \leq f_{Map}(data_{In,1}, \dots, data_{In,k});$$

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 booleschen 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

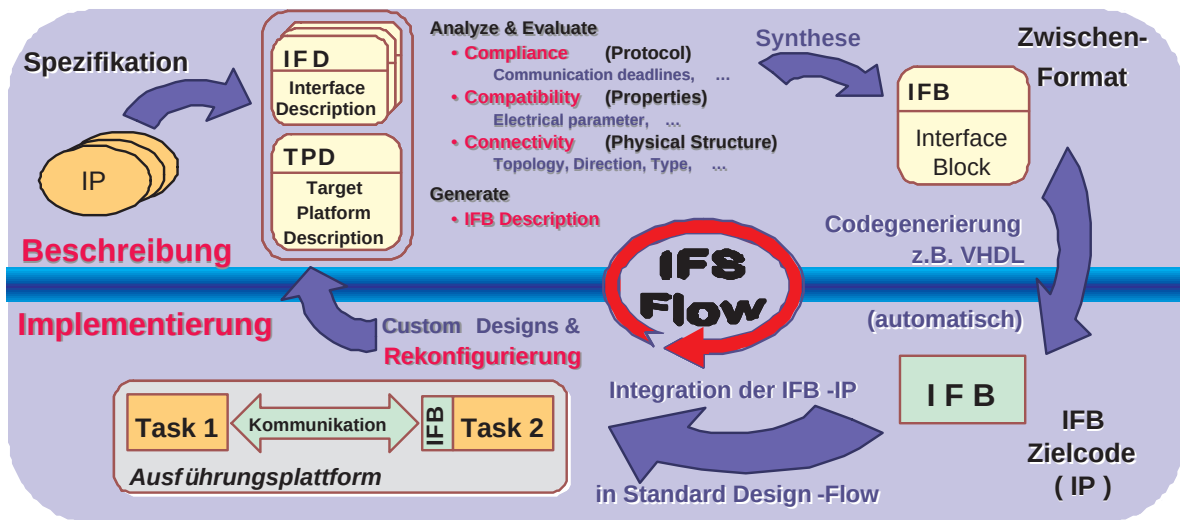


Abbildung 2.9.: IFS-Flow

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.

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 zusätzli-

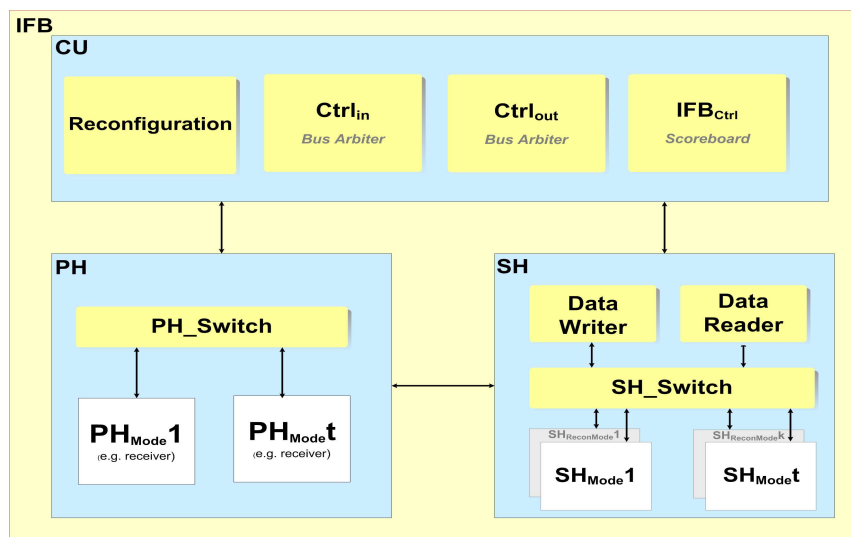


Abbildung 2.10.: IFB-Makrostruktur

2. Grundlagen

che 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 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. Da-

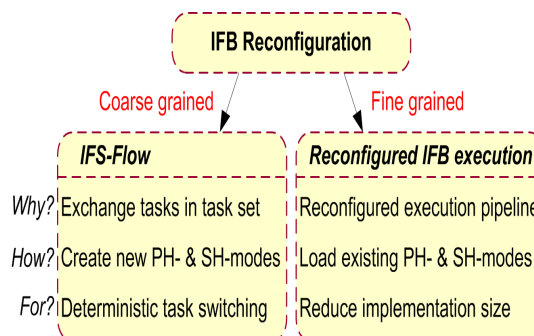


Abbildung 3.1.: Coarse- and Fine grained IFB Rekonfiguration

durch 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 deterministisches 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.

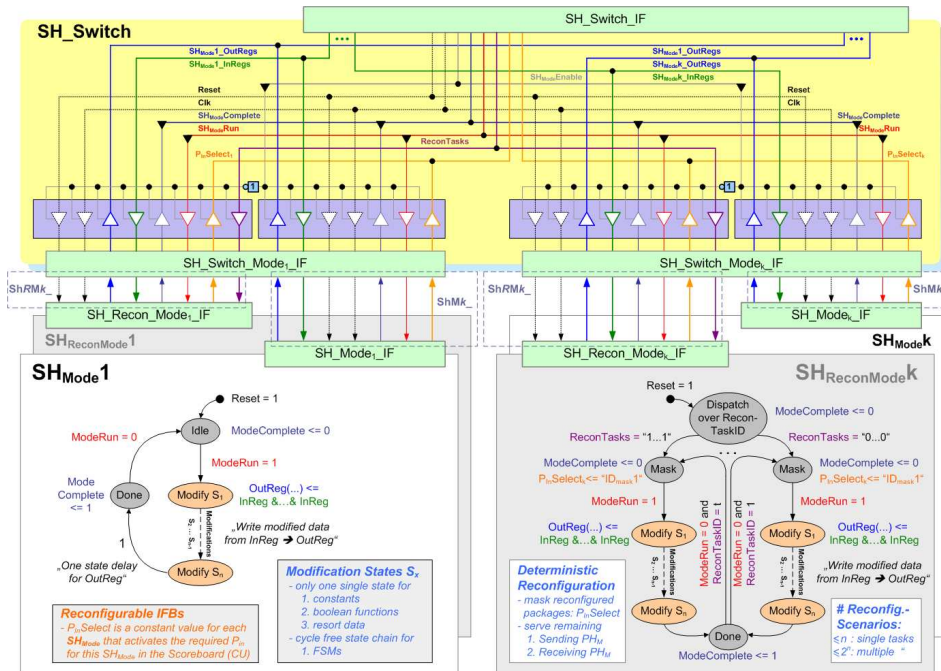


Abbildung 3.2.: SH mit einem Switch, einem Mode und zugehörigem Recon-Mode

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 Schemulern $Ctrl_{In}$ 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 $P_{Inselect}$. Da die Anzahl von Datenpakete variabel ist, ermöglicht $P_{Inselect}$ 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.

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

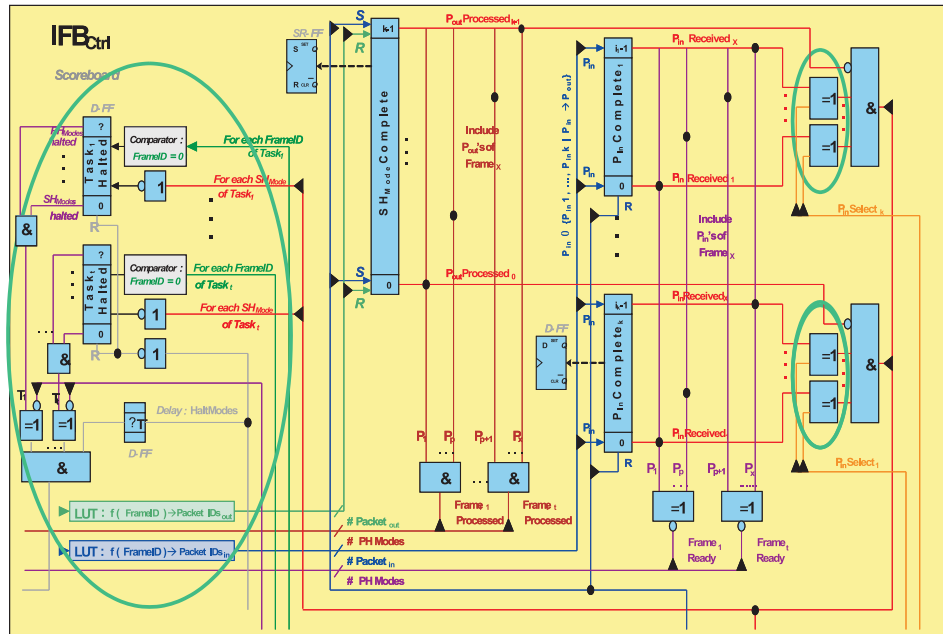


Abbildung 3.3.: Gatterschaltung des Scoreboards in der Control Unit

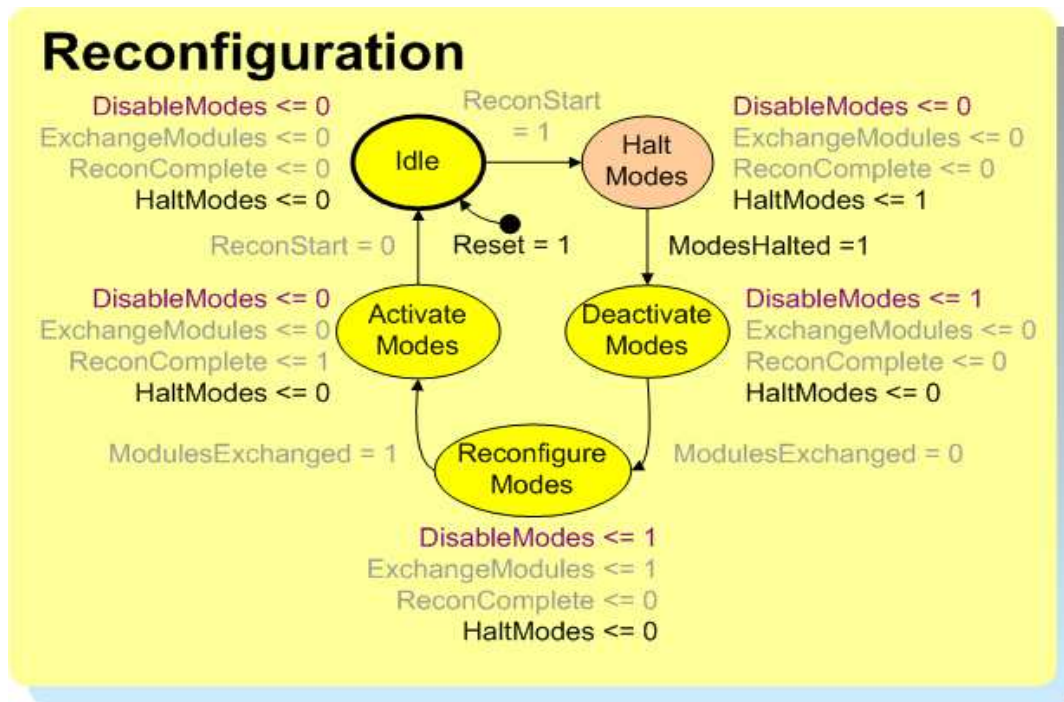


Abbildung 3.4.: Rekonfigurationsablauf in der Control Unit

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.

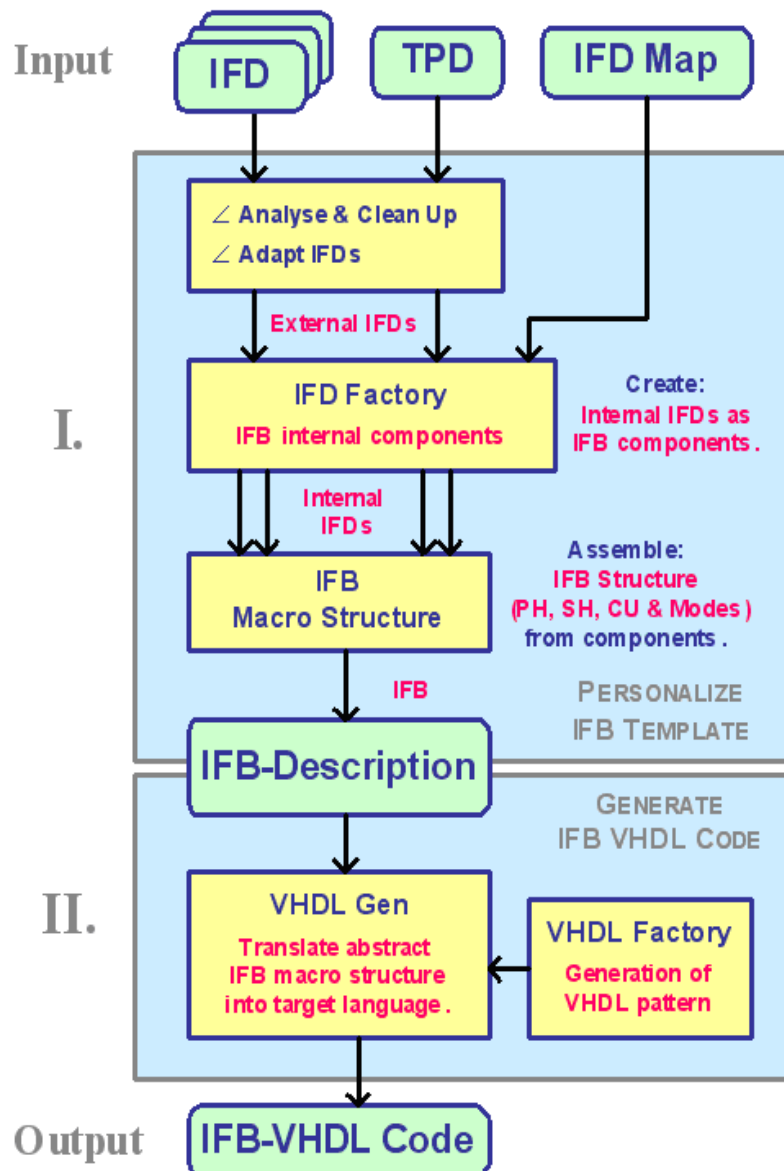


Abbildung 3.5.: Automatisierte IFB Generierung in zwei Syntheseschritten

Im ersten Syntheseschritt werden zuerst die in den Syntheseprozess eingehenden Da-

3. IFB-Synthese

ten 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 Codegenerierung 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 synthesesfähigen 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.

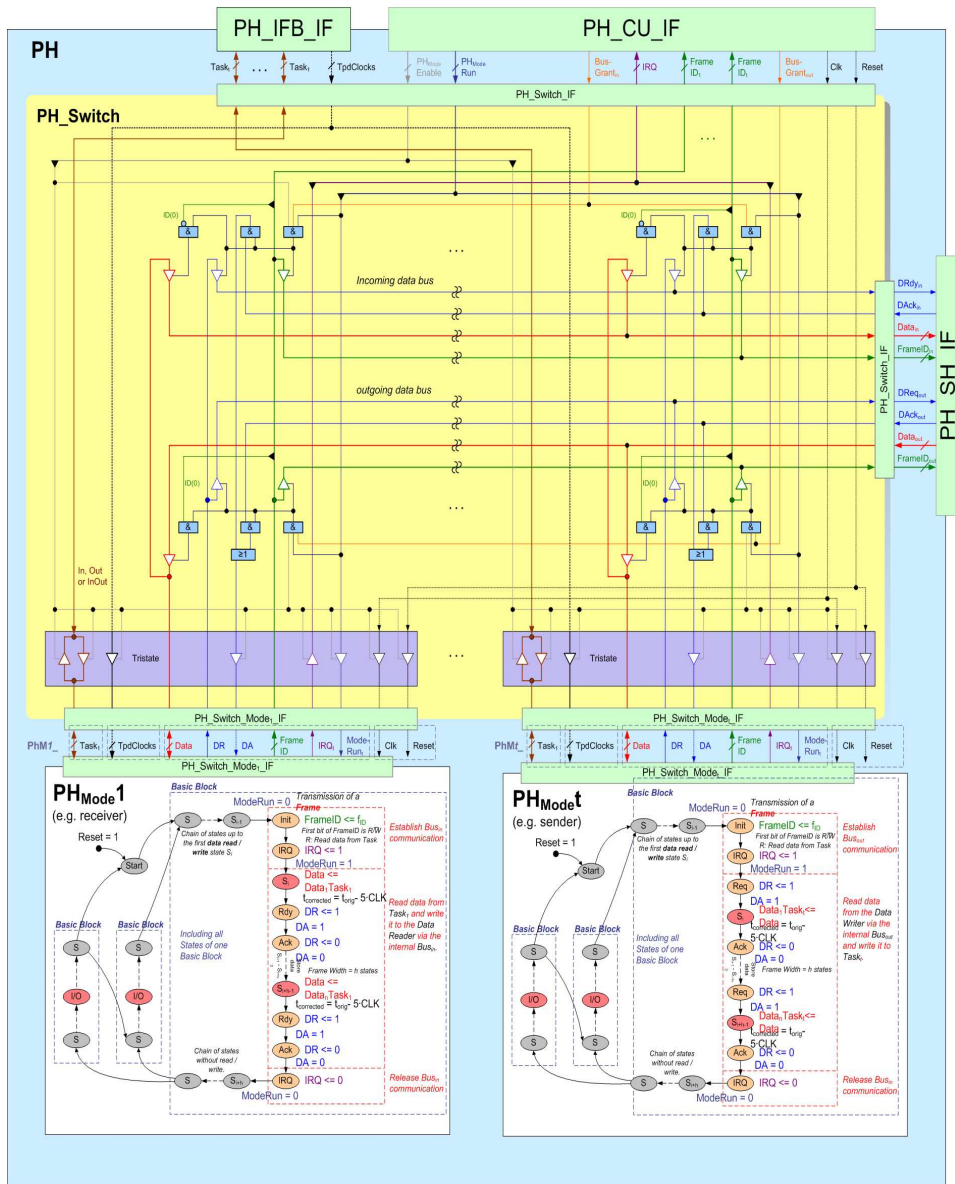


Abbildung 4.1.: Der Protocol Handler und seine Komponenten

4.1.2. Sequence Handler

Wie der Protocol Handler besteht auch der Sequence Handler aus einem data Switch und einer Menge von Modes. Wie im Abbildung 4.2 dargestellt ist, sind zusätzlich ein Data-Reader 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.

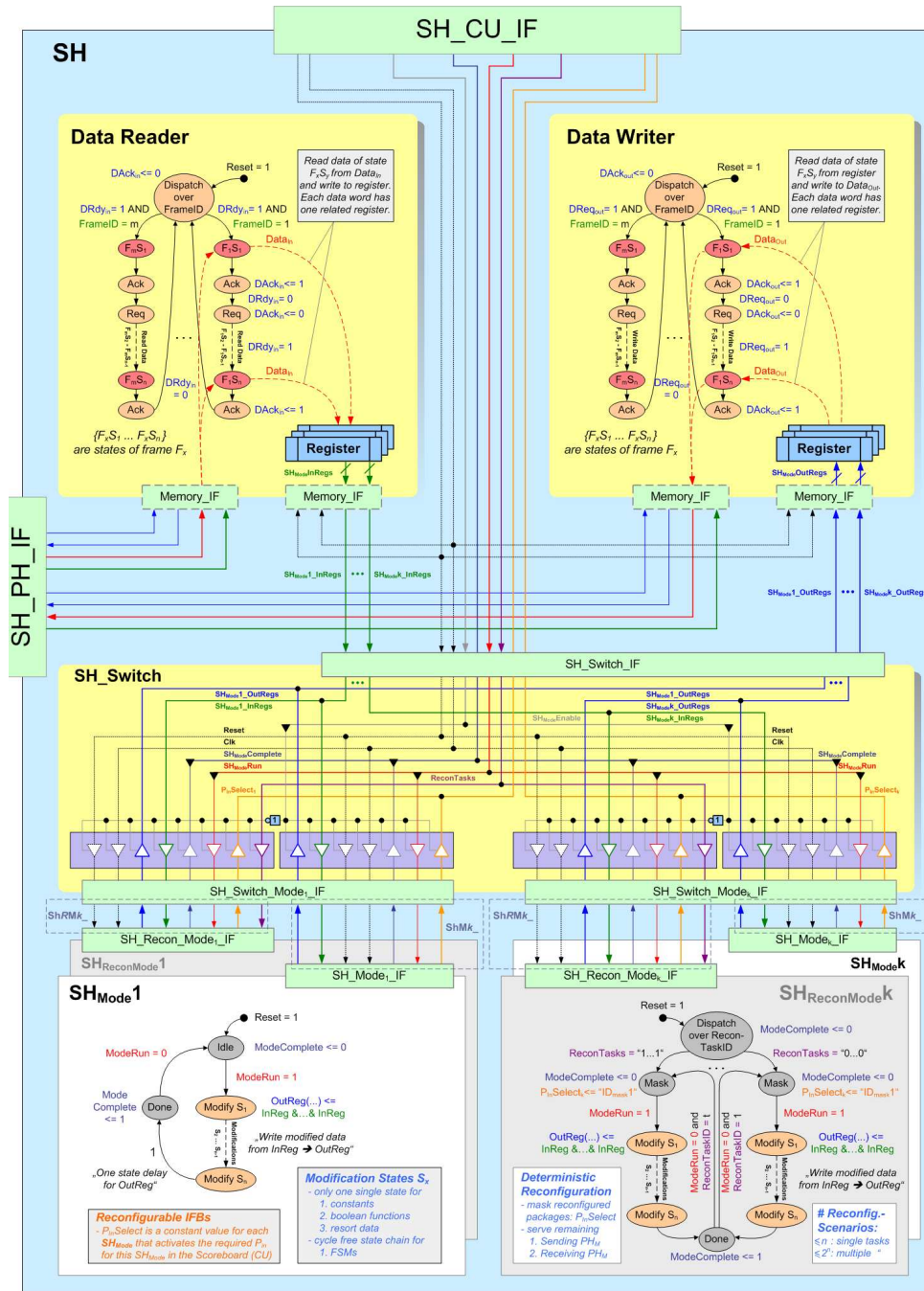


Abbildung 4.2.: Der Sequence Handler und seine Komponenten

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].

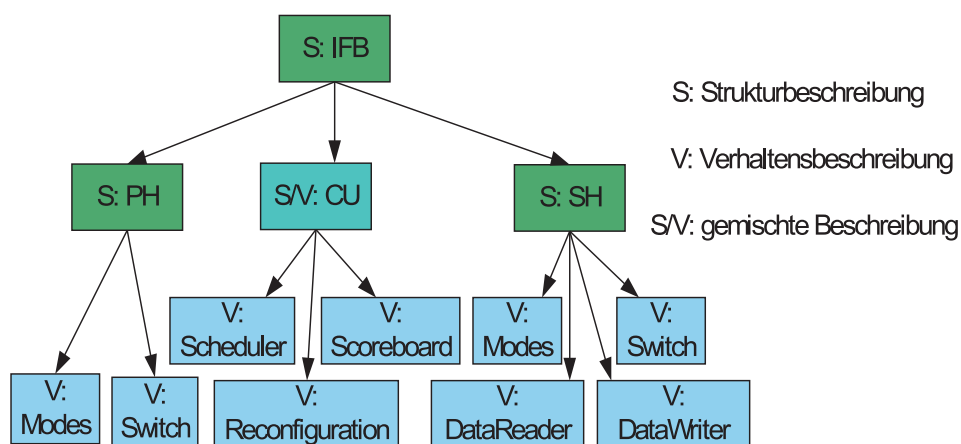


Abbildung 4.3.: Verhalten- und Strukturbeschreibung einer Architektur

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 Synthesalgorithmus, 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.

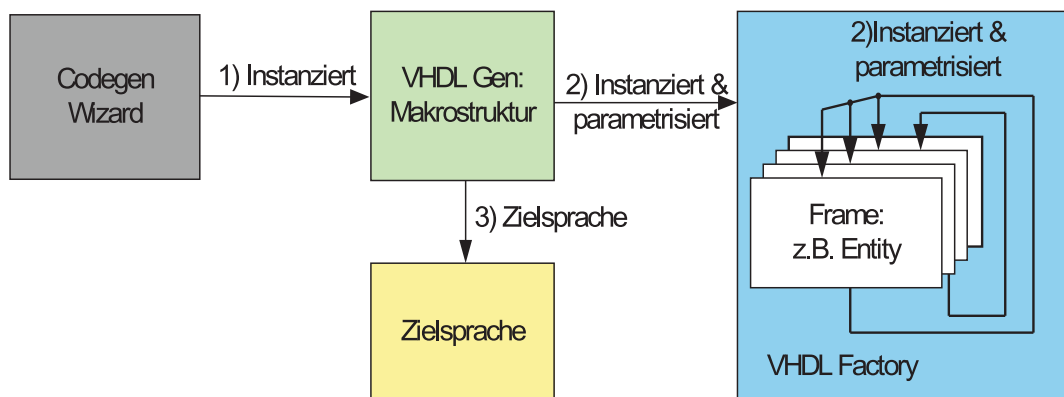


Abbildung 4.4.: VHDL-Codegenerierung durch Frame Processing

4.3.1. VHDL Gen

Wie in 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, Signalfflussrichtung und Signalfbreite 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;
        ...
    end case;

    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;
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 Zeitangaben 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(
```

4. Codegenerierung

```
        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.

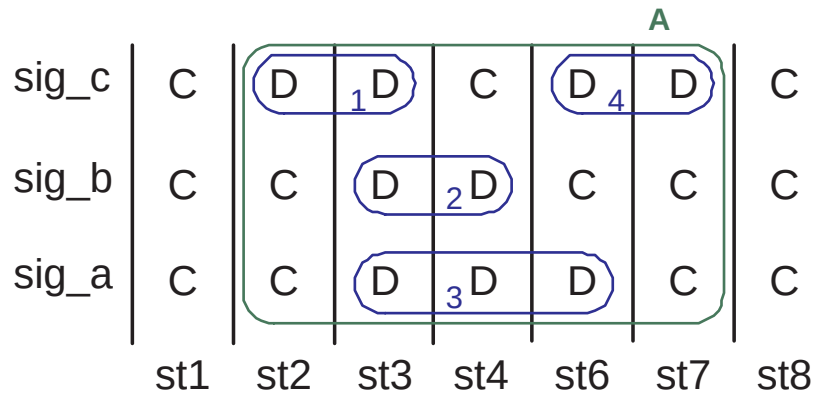


Abbildung 4.5.: Protocol-Matrix

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);
```

4. Codegenerierung

```
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 synthesesfähiger 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

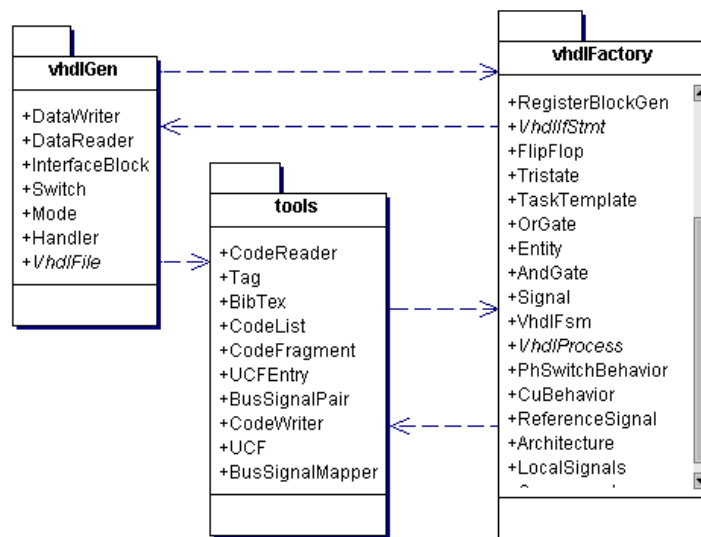


Abbildung 5.1.: Codegenerator

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. vhdGen

Abbildung 5.2 zeigt das Package vhdGen. Für jede Komponente der IFB-Makrostruktur existiert eine Klasse in diesem Package. Neben diesen Klassen beinhaltet vhdGen 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.

5.2.2. vhdFactory

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 vhdFactory 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

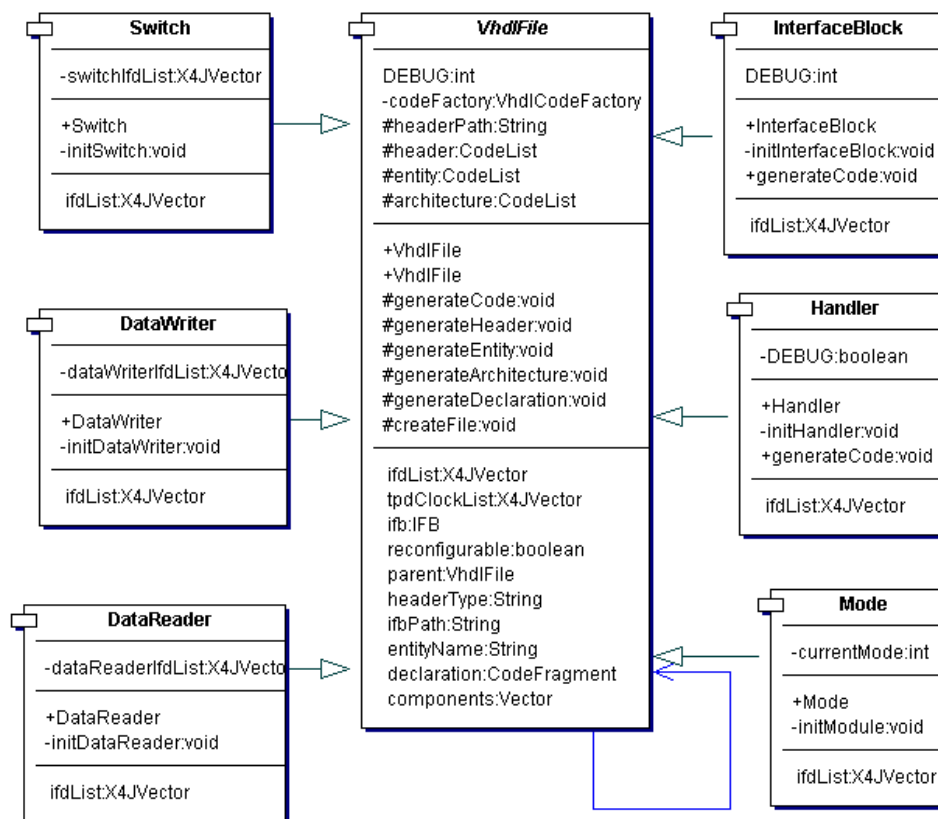


Abbildung 5.2.: Package VhdlGen

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.

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 *Code-Fragment* bezeichnet. Um VHDL-Beschreibungen aus mehreren `CodeFragments` zu erzeugen, müssen diese in der richtigen Reihenfolge zusammengesetzt werden. Aus diesem

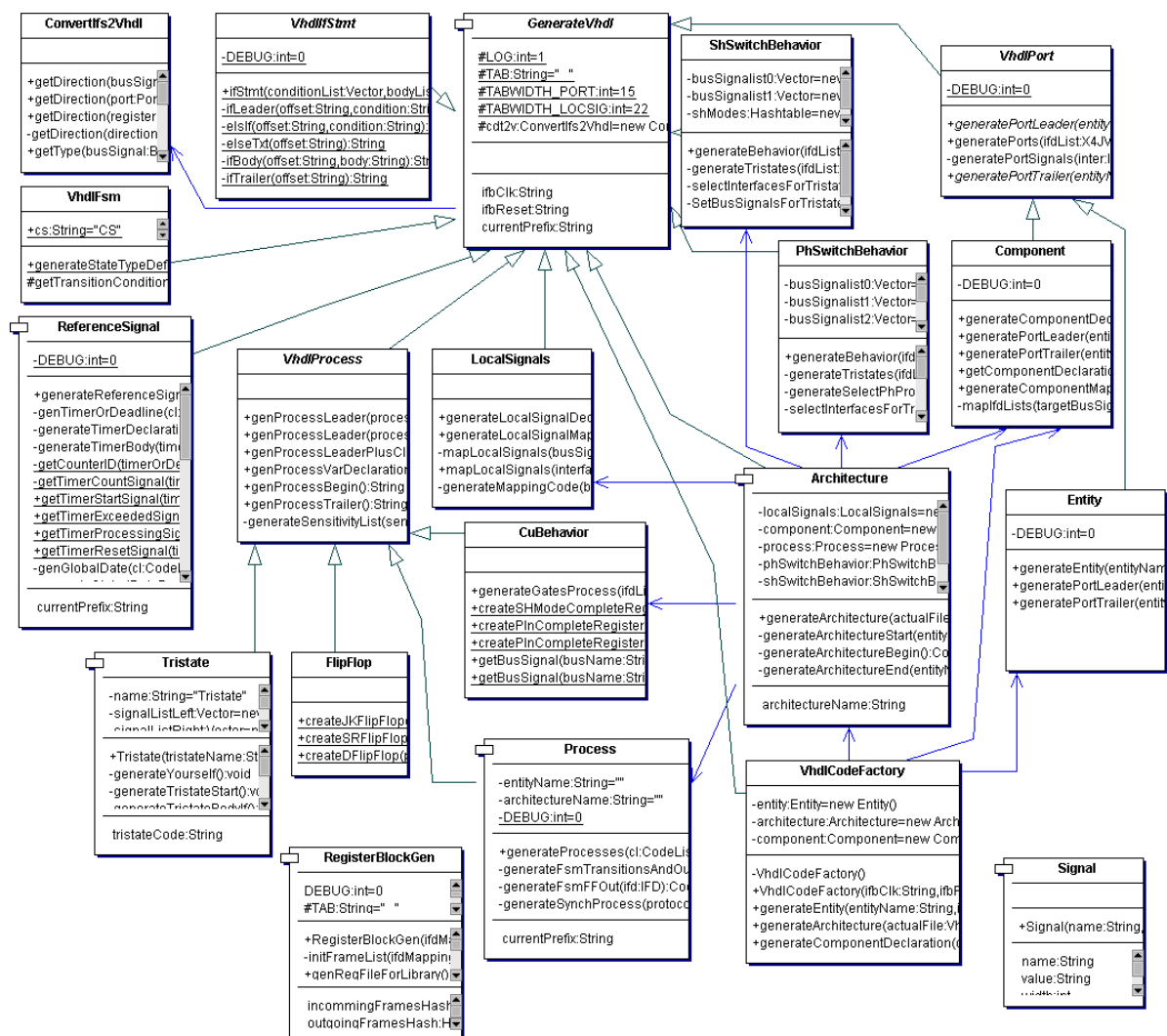


Abbildung 5.3.: Package VhdlFactory

5. Implementierung des Codegenerators

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.

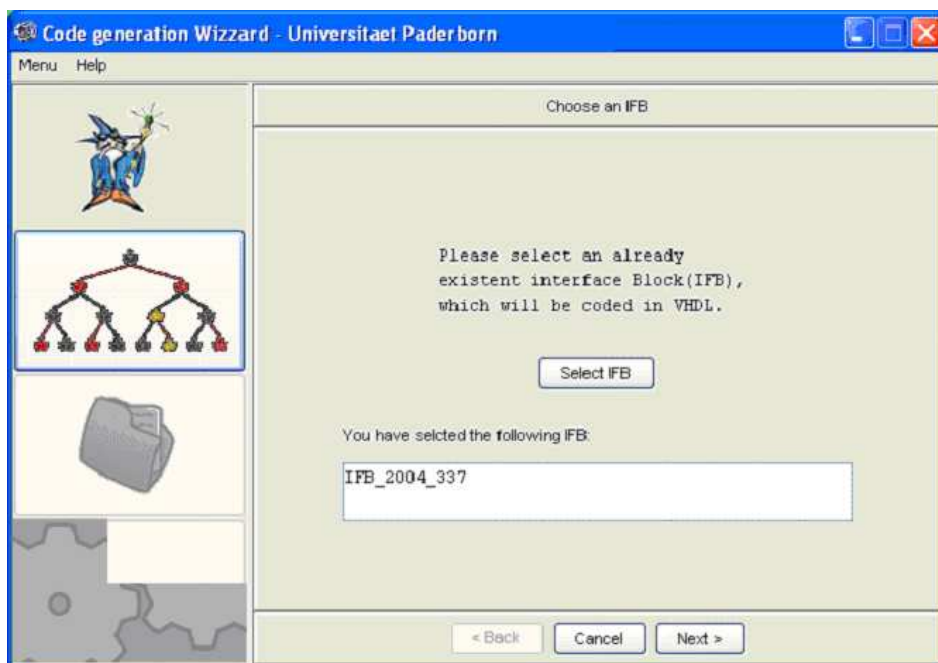


Abbildung 5.4.: CodegenerationWizard, Schritt 1 - Auswahl des IFBs



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

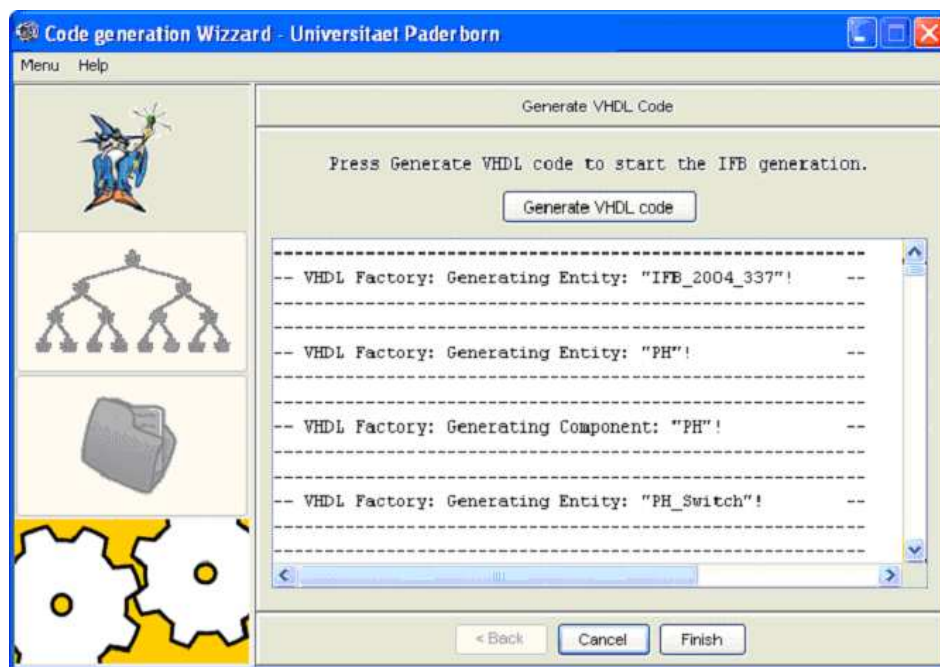


Abbildung 5.6.: CodegenerationWizard, Schritt 3 - Generierung des VHDL-Codes

Abbildung 5.7 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

5. Implementierung des Codegenerators

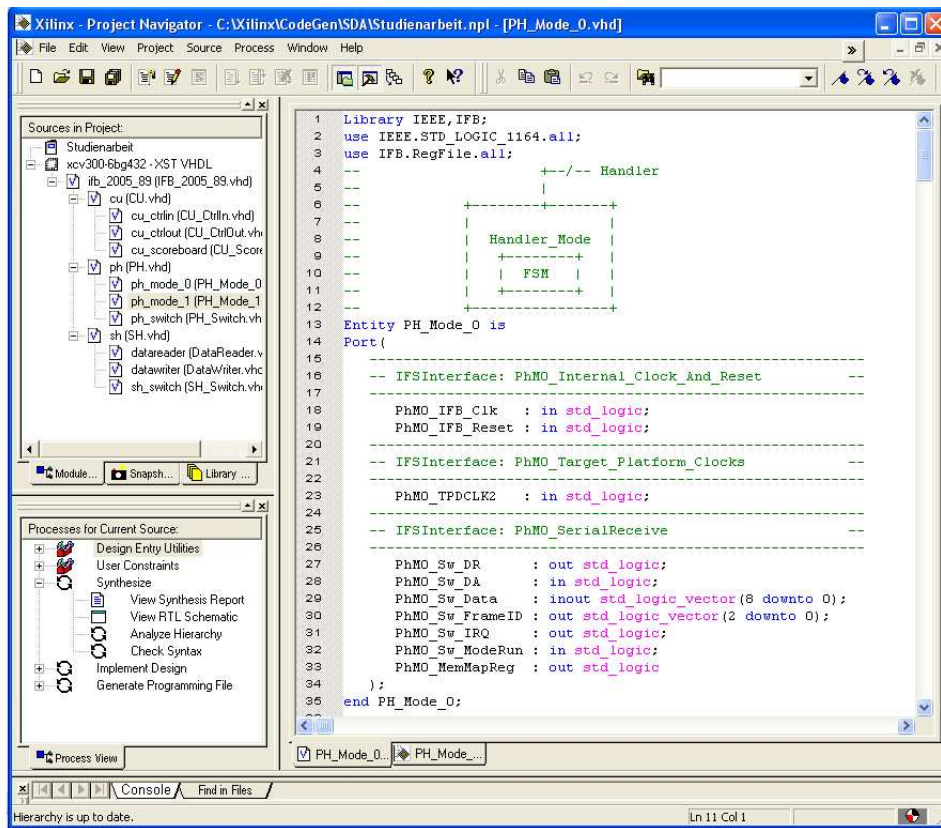


Abbildung 5.7.: Xilinx ISE 6.1 - Ausschnitt aus einer generierten VHDL-Beschreibung

Xilinx.

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 synthesesfähiges 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 *SynthesisWizard* 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.

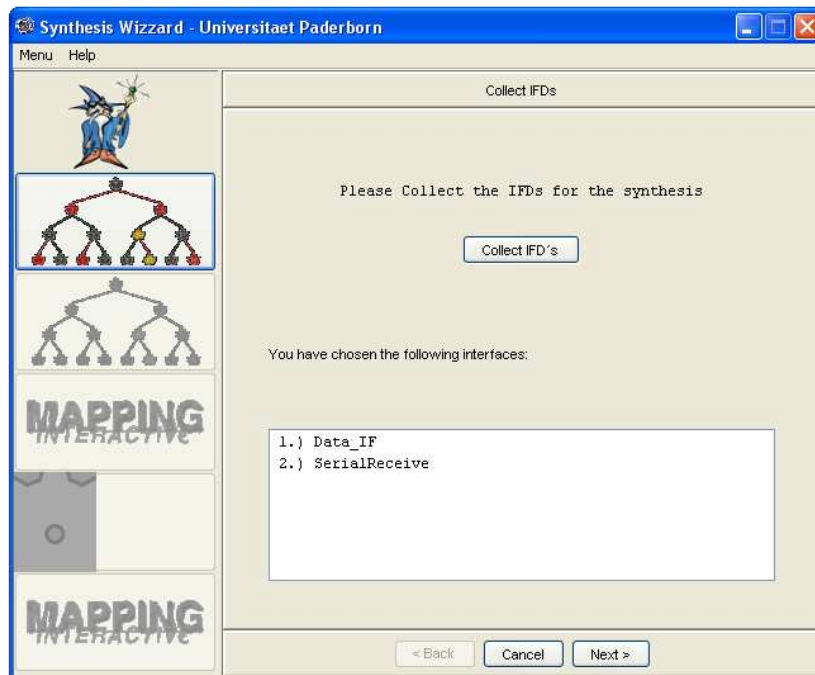


Abbildung A.1.: Synthesis Wizard, Schritt 1 - Auswahl der zu verbindenden Schnittstellen



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

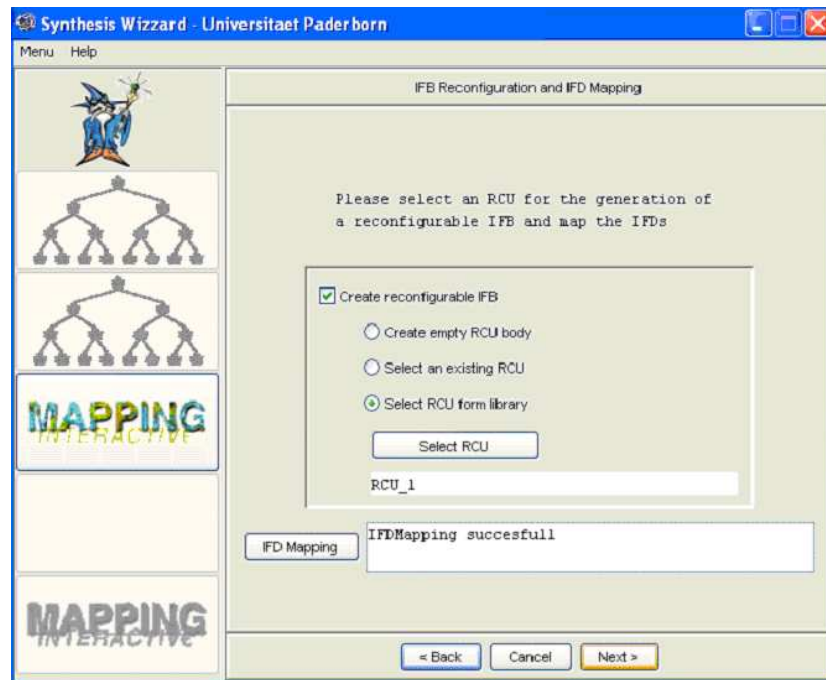


Abbildung A.3.: Synthesis Wizard, Schritt 3 - Definition der Datenabbildung und Festlegung der Rekonfigurierbarkeit des IFBs

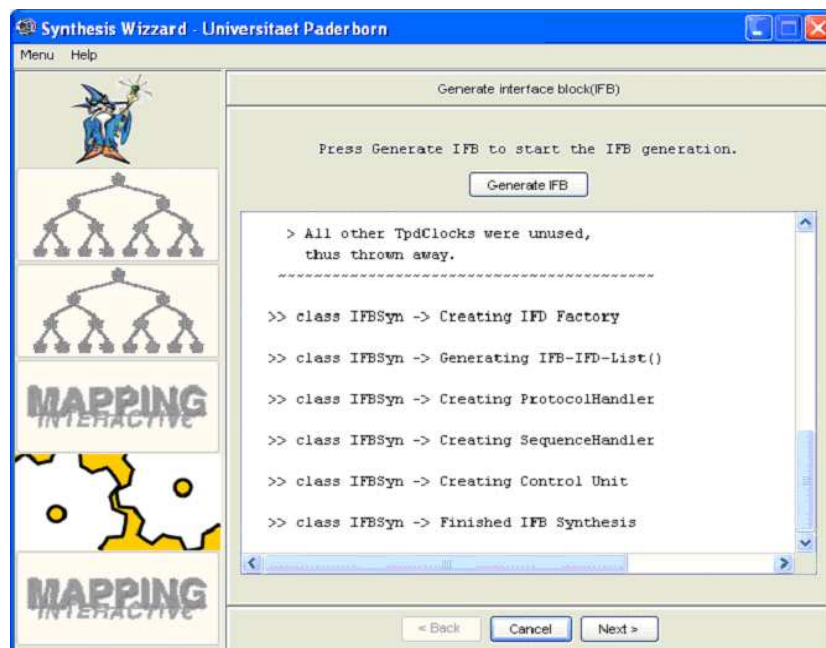


Abbildung A.4.: Synthesis Wizard, Schritt 4 - Erzeugung des IFBs

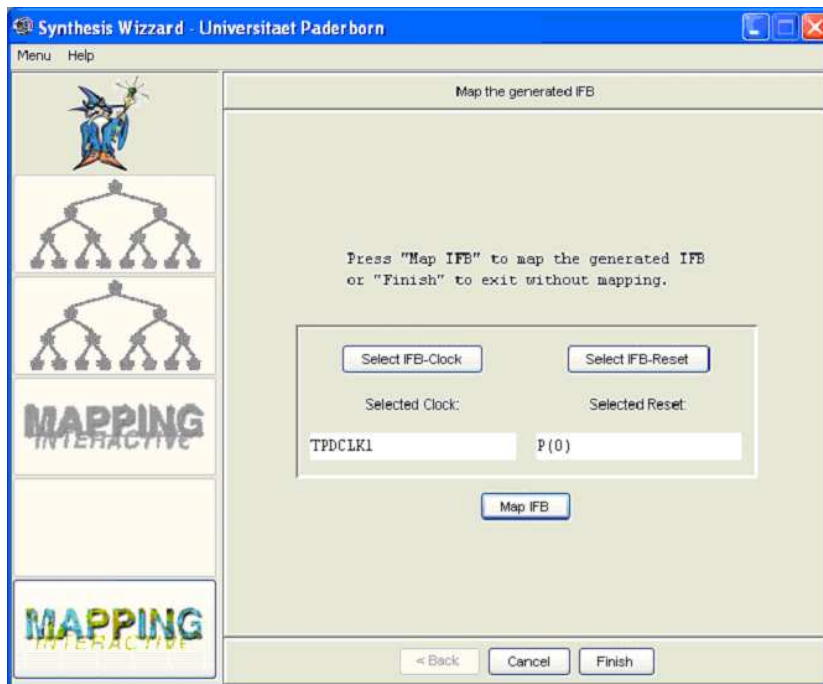


Abbildung A.5.: Synthesis Wizard, Schritt 5 - Einbinden des IFBs

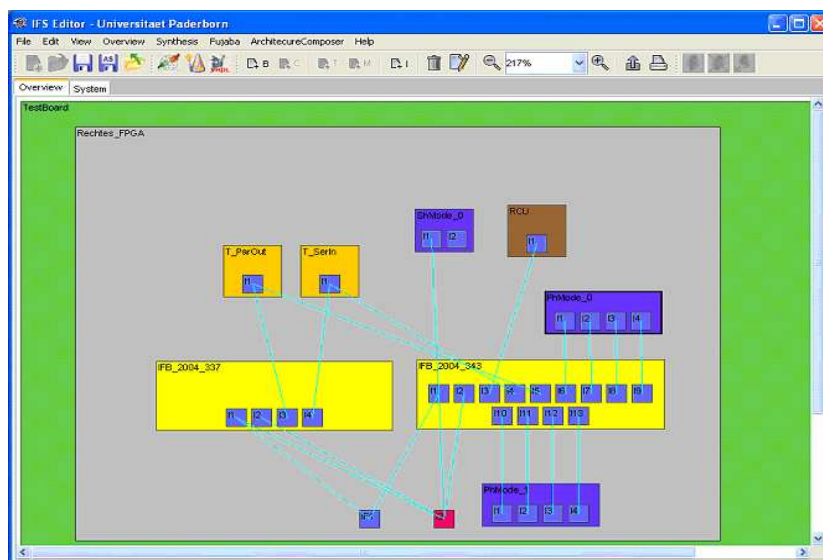


Abbildung A.6.: Der generierte IFB im IFS-Editor

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>.