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

Szenariobasierte Datentransformation semantikfreier Kommunikationsprotokolle für die automatische Synthese in Eingebetteten Systemen

Studienarbeit

Studiengang Informatik

von

Alexander Gepting

Blickstr. 67

32791 Lage

betreut durch

Dipl.-Inform. Stefan Ihmor

vorgelegt bei

Prof. Dr. rer. nat. Franz Josef Rammig

im

Juni 2006

Dank und Erklärung

Diese Studienarbeit ist in der Arbeitsgruppe von Prof. Dr. rer. nat. Franz Josef Rammig (HNI) der Universität Paderborn entstanden.

An dieser Stelle möchte ich mich für das interessante Thema der Arbeit bei Prof. Dr. Franz J. Rammig bedanken. Besonderer Dank gilt meinem Betreuer, Stefan Ihmor, für seine tatkräftige Unterstützung währen der Entstehung dieser Arbeit. Weiterhin danke ich meinen Eltern und meiner Freundin für die aufgebrachte Geduld und Verständnis.

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

Paderborn, 14. Juni 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Motivation

Die erste Massenproduktion von *Eingebetteten Systemen* (*embedded systems, ES*) fand am Anfang der 60er Jahre statt. In den darauf folgenden Jahren wurden die zahlreichen Vorteile dieser Technologie erkannt und immer intensiver genutzt. So existiert heute kaum eine Domäne, aus der eingebetteten Systemen wegzudenken wären. Von der Raum- und Luftfahrttechnik bis zur Autoindustrie, von Haushaltsgeräten bis zur Unterhaltungselektronik und Spielzeugindustrie, alle diese und andere Industriezweige sind ohne die Nutzung dieser Technologie nicht mehr vorstellbar. Dennoch wird von Wirtschaftsexperten auch weiterhin ein großes Wachstum und steigende Marktanteile in diesem Gebiet prognostiziert, was sich unter anderem durch die Präsenz von zahlreichen bereits etablierten und neuen Unternehmen widerspiegelt.

Bedingt durch starke Konkurrenz und steigende Anforderungen an die Funktionalität, stehen die Hersteller unter immensem Druck und sind gezwungen die Entwicklungszeiten für die Realisierung der gewünschten Funktionalitäten zu verkürzen. Durch die Wiederverwendung und Komposition von Teilkomponenten bereits existierender und bewährter Lösungen im Entwurfprozess (*IP-based Design*) von neuen Systemen, lassen sich nicht nur die Entwicklungszeiten verkürzen, sondern auch die damit verbundenen Kosten. Die eingesetzten IPs (*intelectual property*) werden indessen nicht nur aus eigener Produktionspalette, sondern oft von anderen Anbietern bezogen.

Die Integration von bereits vorhandenen Komponenten ist daher ein wesentlicher Bestandteil im Entwurfsprozess von eingebetteten Systemen. Die gewonnene Flexibilität durch die Nutzung verschiedener IPs bedeutet aber gleichzeitig einen Mehraufwand bei deren Einbindung in die Kommunikationsstruktur. Die Heterogenität der eingesetzten Komponenten und die fehlende Standardisierung deren Schnittstellen zwingen die Designer oft zu Kompromissen. Die Herausforderung besteht dabei in der Auswahl verschiedener einzusetzender Komponenten einerseits und Erweiterung ihrer Schnittstellen für die Integration andererseits.

Die Einschränkung bei der Wahl der IPs führt unweigerlich zur Einschränkung der Flexibilität im Entwurfsprozess, erleichtert jedoch die Einbindung der einzelnen Komponente in die Gesamtkommunikationsstruktur. Mit einer steigenden Anzahl von zu integrierenden IPs steigt oft auch der Aufwand für die Erweiterung der dafür notwendigen Schnittstellen. Dieses Vorgehen erweist sich häufig als sehr fehleranfällig und unproportional

aufwendig in Relation zu der gewonnenen Funktionalität.

Um dem Systemdesigner eine Lösung für die Problematik der Einbindung heterogener IPs in ein Gesamtsystem anzubieten, wurde der Ansatz der *Interface Synthesis* (*IFS*) entwickelt [5]. Das Ziel der IFS ist es, durch die automatische Synthese von Schnittstellenadaptern, dem Entwickler eine einfache Möglichkeit zu bieten die – sonst inkompatiblen – Komponenten in ein Gesamtsystem zu integrieren, ohne Änderungen an den Komponenten selbst vornehmen zu müssen. Hierfür werden sogenannte Adaptermodule (*interface block, IFB*) erzeugt, die als "Übersetzer" zwischen inkompatiblen Komponenten eingesetzt werden und somit einen Informationsaustausch ermöglichen.

Der Ablauf des IFS-Ansatzes ist in vier Schritte gegliedert, dem sogenannten *IFS-Flow*. Im ersten Schritt erfolgt die Modellierung des Kommunikationssystems mit der Beschreibung der Ausführungsplattform und den miteinander kommunizierenden Komponenten. Dies beschreibt die sogenannte Kommunikationsinfrastruktur. Die Schnittstellenbeschreibungen (*interface description, IFD*), welche unter anderem das Protokoll der Kommunikationskomponenten (z.B. *IPs*) beinhalten, müssen wahlweise entweder von Systemdesigner oder – falls vorhanden – von dem Hersteller der IP (z.B. in mitgelieferten Bibliotheken) in einem XML-Format (sog. IFS-Format) bereitgestellt werden.
Der Modellierungsphase folgt im zweiten Schritt die automatische Synthese einer abstrakten und zielsprachenunabhängigen IFB-Beschreibung. Diese wird anschließend für die Generierung einer Implementierung des IFBs in einer konkreten Zielsprache eingesetzt, und kann dann, im letzten Schritt des IFS-Flows, in das existierende Design integriert werden.

Der Aufbau des IFB basiert auf der sogenannten IFB-Makrostruktur. Diese arbeitet nach dem Eingabe-Verarbeitung-Ausgabe (EVA) Schema und besteht aus zwei Protocol Handlern, einem Sequence Handler und der Control Unit. Die Protocol Handler übernehmen die Kommunikation mit den – aus der Sicht des IFB – externen IPs und leiten die Informationen zur Weiterverarbeitung an den Sequence Handler weiter. Gesteuert wird dieser Transformations- und Verarbeitungsprozess von der Control Unit.

Die Informationen aus den Schnittstell- und Protokollbeschreibungen der verwendeten Kommunikationskomponenten sagen nichts über deren Bedeutung aus. Jedoch wird für die Datenverarbeitung innerhalb des IFB, genauer gesagt für Transformation der Daten von Sender zum Empfänger, die gemeinsame Semantik benötigt um die Nutzdaten des eingehenden Protokolls auf das Ausgehende automatisch abbilden zu können. Die Bedeutung der Daten sowie die gewünschte Abbildung dieser zwischen den Kommunikationskomponenten ist im Allgemeinen nur dem Designer des Systems bekannt. Somit lässt sich der Schnittstellenadapter (IFB) zwischen zwei Kommunikationspartnern erst nach der vorherigen Definition der Abbildungsvorschriften automatisch synthetisieren.

1.2 Aufgabenstellung

Im Rahmen dieser Studienarbeit soll der *Interface Synthesis Design Flow* um die Möglichkeit der automatischen Datentransformation erweitert werden. Aufbauend auf dem bestehenden Datenformat für die Beschreibung von Schnittstellen, welches lediglich die Protokollsyntax spezifiziert, ist eine für die automatische Protokolltransformation benötigte Protokollsemantik in Form von Abbildungsvorschriften (*IFD-Mapping*) zu entwickeln. Diese soll es erlauben, die in der Protokollsyntax vorgegebenen Daten szenarioabhängig aufeinander abzubilden.

Für die Formulierung der Abbildungsvorschriften ist eine Grammatik zu definieren, die folgende Operationen für die Abbildungen von Daten unterstützt:

- Zuweisung konstanter Werte
- Zuweisung eingehender auf ausgehende Bits (ohne diese zu verändern)
- Anwendung von booleschen Funktionen auf eingehende Bits
- Verwendung eines endlichen Zustandsautomaten (FSM) zur Datenmanipulation

Als Grammatik beschrieben ist die Abbidungssprache in den bestehenden Syntheseablauf des IFS-Editors zu integrieren, so dass die eigentliche Schnittstellensynthese voll automatisch ablaufen kann. Die aus der Abbildungssprache gewonnenen Informationen sind dabei innerhalb der Synthese zur Personalisierung und der Optimierung des erzeugten Schnittstellenadapters anzuwenden. Hierbei ist aufzuzeigen, in wieweit die Implementierung des Sequence Handlers und die szenariobasierte Ausprägung der Control Unit durch die Datenabbildung bestimmt wird. Weiterhin ist die Optimierung des Protocol Handlers basierend auf der Datenabbildung zu beschreiben.

Die Abbildungssprache ist als Datenstruktur (*IFD-Mapping*) zu implementieren. Das *IFD-Mapping* liefert neben der Beschreibung der Ausführungszielplattform (*target platform description, TPD*) und der Schnittstellenbeschreibung (*IFD*) die erforderliche dritte Eingabe für die automatische schnittstellensynthese im IFS-Flow.

1.3 Aufbau der Arbeit

Diese Arbeit ist in fünf Kapiteln gegliedert. Nach der Einführung folgt ein Kapitel über die Grundlangen und die bereits existierenden Konzepte. Es werden die zentralen Begriffe der IFS und aus dem Bereich der formalen Sprachen erklärt, auf denen diese Arbeit aufbaut. Hierzu wird der Terminus Grammatiken kurz erläutert und anhand von Beispielen verdeutlicht. Formalisiert wird der Begriff durch Definition von Grammatiken und deren Klassifizierung in verschieden Typen der Chomsky-Hierarchie. Als wichtiger Bestandteil und Voraussetzung für die Kapitel drei bis fünf, wird hier auf die Begrifflichkeiten wie Compiler, IFS-Flow, IFB und Protokolle genauer eingegangen.

In Kapitel drei wird das Konzept des IFD-Mappings als Menge von Abbildungsvorschriften vorgestellt. Unter Zuhilfenahme von Beispielen wird ferner die Grammatik der Sprache entwickelt, die für die Beschreibung der einzelnen Datenabbildungen zwischen inkompatiblen Protokollen eingesetzt wird. Anschließend werden die Einschränkungen und Grenzen für die Anwendung des entwickelten Verfahrens im Rahmen der Schnittstellensynthese aufgezeigt.

Kapitel vier zeigt die Auswirkungen des IFD-Mappings auf die Makrostruktur des Interface Blocks (IFB). Unter anderem werden die Optimierungsmöglichkeiten hinsichtlich des Platzbedarfs und Zugriffszeiten auf die interne Busstruktur des IFB besprochen. Der Schwerpunkt liegt dabei auf der Generierung der SH-Modes im SH, sowie auf der Parametrisierung der IFB- Komponenten hinsichtlich eines Anwendungsszenarios.

In Kapitel fünf wird die praktische Umsetzung des IFD-Mapping in Java und dessen Integration in den IFS-Editor vorgestellt. Hierfür werden der entwickelte Mapping-Editor und die wichtigsten Java-Klassen mit ihren Methoden präsentiert. Anschließend wird kurz auf die Generierung des syntaktischen und semantischen Analysierers aus der entwickelten Grammatik mit Hilfe von JavaCC und JJTree eingegangen.

Im letzten Kapitel sechs wird eine Zusammenfassung der vorgestellten Verfahren mit einem Ausblich auf mögliche Erweiterungen gegeben.

2 Grundlagen

2.1 Interface Synthesis Design Flow (IFS-Flow)

Der *Interface Synthesis Design Flow* (IFS-Flow) ist eine Methodik für die Modellierung und die automatische Synthese von rekonfigurierbaren Schnittstellen in eingebetteten Systemen. Die flexible Wiederverwendung und Integration von inkompatiblen IPs (*intellectual property*) war ein zentraler Bestandteil bei der Entwicklung des IFS-Flows. Hierfür bietet dieses Verfahren die Möglichkeit für die automatisierte Synthese von Schnittstellenadaptern für inkompatible IPs [6]. Der IFS-Flow setzt dabei keine tiefgehenden Kenntnisse über die eingesetzten IPs voraus, was den Wunsch nach Geheimhaltung des inneren Aufbaus und der Funktionalität seitens der Hersteller unterstützt (*black box*). Für die Integration von IPs in das modellierte System wird nur eine Schnittstellenbeschreibung (*Interface Description, IFD*) benötigt. Diese Eigenschaft ermöglicht einen flexibleren und schnelleren Entwurf von Prototypen.
Die Grundlage für den Entwurfprozess nach dem IFS-Flow bildet eine abstrakte Beschreibung der Systemarchitektur.

Systemarchitektur

Für die Beschreibung komplexer Kommunikationsszenarien (*System Prototyping, IP-based Design*) wurde das Model der *IFS-Systemarchitektur* entwickelt (siehe Abbildung 2.1). Sie bildet eine hierarchische Struktur aus miteinander verbundenen *Systemkomponenten* und ermöglicht es die gewünschte Kommunikationsstruktur zu modellieren.
Die in der IFS-Systemarchitektur eingesetzten Systemkomponenten werden in *Architekturkomponenten* und *Kommunikationskomponenten* unterschieden. Zu den ersteren zählen die Komponenten *System, Board* und *Chip*. Dagegen gehören zu den *Kommunikationskomponenten* die *Tasks* und die *Medien*. Die miteinander kommunizierenden Komponenten lassen sich nicht nur auf einer Architekturkomponente (*System-On-Chip, SoC*) platzieren, sondern können auf beliebige andere verteilt und über deren Schnittstellen miteinander verbunden werden. Im Gegensatz zu den Medien unterliegen die Tasks einer Einschränkung. Sie dürfen ausschließlich auf einem Chip positioniert werden.

Beide Arten von Komponenten verfügen über Schnittstellenbeschreibungen (*Interface Description, IFD*). Die Kommunikationskomponenten besitzen zusätzlich zur strukturellen Beschreibung der Schnittstellentopologie eine Verhaltensbeschreibungen in Form von Protokoll-Zustands-Automaten (*finite state machines, FSM*). Kompatible Kommu-

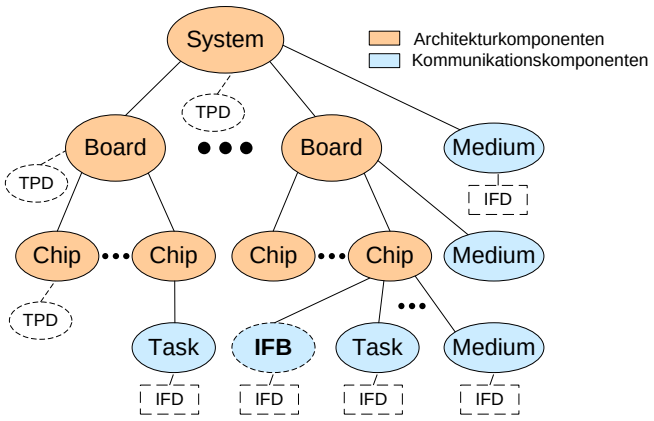

Abbildung 2.1: IFS Systemarchitektur

nikationskomponenten können über die in der *Systemarchitektur* vorhandenen Verbindungen kommunizieren. Inkompatible Tasks und Medien werden hingegen durch den so genannten *Interface Block* (*IFB*) verbunden. Dazu später mehr.

Beschrieben wird die Systemarchitektur in dem *Interface Synthesis Format* [1]. Als XML-Schema beschreibt dieses Format die physikalische Struktur, die Eigenschaften und die Protokolle aller Systemkomponenten.

Target Platform Description (TPD)

Die Beschreibung der Zielplattformeigenschaften im IFS-Flow wird als *Target Platform Description* (*TPD*) bezeichnet. Sie repräsentiert die Komponente des Systems, auf der der synthetisierte IFB ausgeführt werden soll. Ihre Eigenschaften, wie die verfügbaren Ressourcen und vorhandenen Clocks der Zielplattform, werden entsprechend bei der Synthese berücksichtigt.

Interface Description (IFD)

Mit Hilfe von *Interface Descriptions* (*IFDs*) werden die Schnittstellen der Systemkomponenten beschrieben. Neben der *TPD* repräsentiert die *IFD* die Eingabe mit statischen Informationen für die automatische Schnittstellensynthese, und muss entweder von dem Designer des Systems oder dem Hersteller der Komponente bereitgestellt werden. Der Inhalt der IFD sind entwicklerabhängige, jedoch von dem Anwendungsszenario unabhängige Informationen, so dass die IFDs im IP-based Design beliebig eingesetzt werden können. Die IFD kann als eine Art Container angesehen werden, die durch ein Tripel (I-P-M) definiert wird, wobei die einzelnen Elemente folgende Bedeutung haben [4]:

- Ein *Interface (I)* beschreibt eine Menge von physikalischen Pins mit deren Attributen wie Richtung, Bitbreite, Typ usw.

[1] Die Beschreibung des Systems mit ihren Komponenten im IFS-Format lässt sich mit Hilfe des Java-Tools "IFS-Editor" erzeugen.

- Ein *Protocol (P)* definiert eine Menge von virtuellen Protokollpins sowie deren Verhalten, das durch Protokollzustandsautomaten beschrieben wird. Die Ausgaben des Automaten legen die Werte der einzelnen Pins in den jeweiligen Zuständen fest.
- Eine *ProtocolMap (M)* beschreibt eine Abbildung der virtuellen Protokollpins auf die physikalischen Pins eines Interfaces.

IFS-Flow

Der *Interface Synthesis Design Flow* besteht aus vier Schritten (siehe Abbildung 2.2).

1. *Modellierung der Systemarchitektur*
   Ist die vollständige Beschreibung des Systems mit allen benötigten IPs vorhanden, wählt der Designer die Schnittstellenbeschreibungen (IFD) der Tasks und Medien aus, die verbunden werden sollen, sowie die Zielplattformbeschreibung (TPD) der Komponente, auf der der IFB ausgeführt werden soll. Da die Bedeutung der Daten der IPs nicht bekannt ist, werden zusätzlich dynamische Informationen (siehe Kapitel 3) für den Datenaustausch zwischen den Kommunikationskomponenten benötigt. Diese müssen vom Designer des Systems vor der Synthese erstellt oder, falls bereits vorhanden, in Form einer Bibliothek bereitgestellt werden. Die IFDs werden auf Einhaltung bestimmter Protokolleigenschaften (z.B. Deadlines), Kompatibilität (z.B. elektrische Parameter des Interfaces) und Konnektivität (z.B. Typ und Richtung der Protokollpins) überprüft. Werden alle notwendigen Bedingungen erfüllt, kann eine einfache, direkte Verdrahtung zwischen den ausgewählten Komponenten erstellt werden. Andernfalls wird ein Interface Block (IFB) als Adaptermodul synthetisiert. Eine Voraussetzung dafür ist jedoch, dass die Signale elektrisch kompatibel sind [4].

2. *Syntheseschritt 1*
   Im ersten Schritt der Synthese wird einer abstrakte Instanz des IFB in einem Zwischenformat erstellt. Dieser IFB kann dann in Form einer XML-IP gespeichert, importiert und exportiert werden.

3. *Syntheseschritt 2*
   Im zweiten Syntheseschritt wird der endgültige Code der IFB-Implementierung in einer Zielsprache, in Abhängigkeit von der angestrebten Lösung, generiert. Die angestrebte Lösung kann eine Software- oder Hardwarelösung sein.

4. *Integration*
   Die so erzeugte IFB-Instanz wird in die bereits existierende Implementierung der modellierten Systemarchitektur integriert.

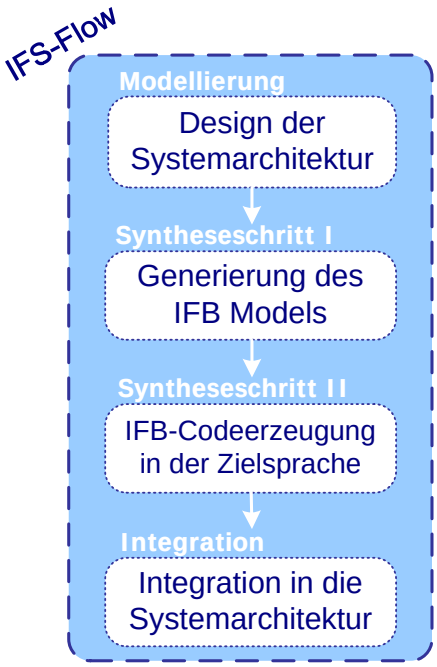

Abbildung 2.2: Interface Synthesis Design Flow

2.2 Der Interface Block (IFB)

Der *Interface Block* (IFB) fungiert, wie bereits im vorherigen Kapitel angesprochen, als ein Adapter-Modul und wird durch den IFS-Flow dynamisch erzeugt. Mit seiner Unterstützung lassen sich Kommunikationspartner (IPs) verbinden, deren Kommunikationsstruktur nicht kompatibel ist, und somit generell keine Verbindung möglich wäre, ohne Änderungen an den IPs vornehmen zu müssen. So können zum Beispiel zwei Tasks mit verschiedenen Protokollen dennoch Daten untereinander austauschen. Der IFB ist dabei für die beiden Kommunikationspartner völlig transparent.
Der konzeptionelle Aufbau eines IFB wird durch die IFB-Makrostruktur beschrieben.

IFB-Makrostruktur

Der interne Aufbau eines IFB wird als *IFB-Makrostruktur* bezeichnet. Die IFB-Makrostruktur wird durch eine Reihe von endlichen Zustandsautomaten (finite state machine, FSM) implementiert [8].

Wie in der Abbildung 2.3 dargestellt, wird der IFB in drei logisch zusammenhängende Hauptkomponenten unterteilt: *Protocol Handler* (PH), *Sequence Handler* (SH) und *Control Unit* (CU). Im Folgenden werden der Aufbau und die Funktionsweise der einzelnen Komponenten kurz erläutert.

Der *Protocol Handler* (PH) dient als Schnittstelle zwischen den externen Tasks bzw. Medien und den anderen Komponenten des IFB. Er beinhaltet eine Menge an Stubs, den

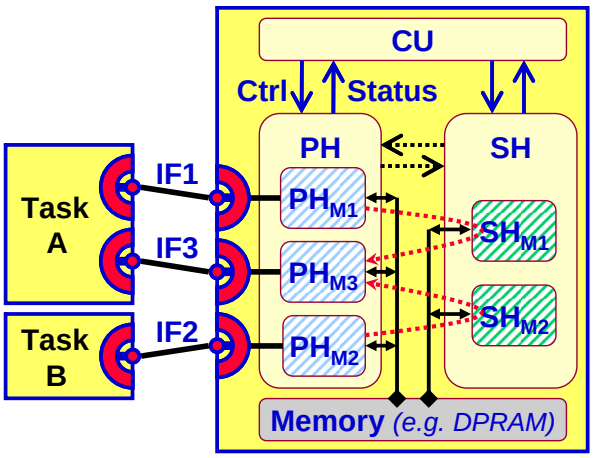

Abbildung 2.3: IFB-Makrostruktur

so genannten *PH-Modes*, die jeweils mit der Schnittstelle einer Task bzw. eines Mediums verbunden sind, um die Kommunikation mit dieser zu ermöglichen. Das Verhalten der einzelnen PH-Modes wird jeweils durch einen endlichen Zustandsautomaten beschrieben, der das komplementäre Protokoll der Task oder eines Mediums realisiert. Daher kann ein PH-Mode nur mit der Task bzw. dem Medium kommunizieren, aus dessen Schnittstellenbeschreibung er in Form einer FSM implementiert wurde. Andererseits wird für jede Task und jedes Medium, die mit dem IFB verbunden werden, ein PH-Mode benötigt. Für die interne Kommunikation mit den anderen Komponenten des IFB wie der CU bzw. dem SH wird ein PH-Mode um zusätzliche Zustände ergänzt.

Der *Sequence Handler* (SH) besteht aus den SH-Modes und dem IFB internen Speicher (*DataReader* und *DateWriter*). Das Verhalten der *SH-Modes* wird ebenfalls durch FSMs beschrieben.
Der SH ist die Komponente, welche die "Übersetzung" der Daten vornimmt. Wie die Daten letztendlich abgebildet werden, wird durch das IFD-Mapping (siehe Kapitel 3) festgelegt. Der DataReader und der DataWriter fungieren dabei als Zwischenspeicher für diese Abbildung bzw. "Übersetzung" der eingehenden und ausgehenden Daten und sind über Datenbusse mit SH-Modes und PH-Modes verbunden.
Die Synthese des Speichers erfolgt aufgrund der Informationen aus dem IFD-Mapping. Dabei entspricht seine Größe genau der Menge der Daten, die empfangen bzw. durch den Transformationsvorgang generiert und versendet werden.

Die Ablaufsteuerung des SH und des PH übernimmt eine weitere Komponente der Makrostruktur des IFB, die *ControlUnit* (CU). Sie ist unter anderem für die Arbitrierung der Datenbusse verantwortlich, indem sie die Steuersignale an den PH und SH sendet und deren Statussignale auswertet. In einem Multitask-IFB – ein mit mehreren sendenden und empfangenden Tasks verbundener IFB – übernimmt die CU auch das Scheduling der Zugriffe einzelner Komponenten auf die IFB internen Ressourcen.
Zu diesem Zweck implementiert die CU zwei Scheduler, jeweils einen für das Memory-

interface des eingehenden bzw. ausgehenden Speichers. Dazu gibt der erste Scheduler den entsprechenden Bus im IFB frei, über den der für den Datenempfang zuständige PH-Mode diese in den Zwischenspeicher (Data Reader) schreibt. Der zweiter Scheduler entscheidet entsprechend welcher PH-Mode die Daten über den Data Writer versenden soll und verwaltet entsprechend den Zugriff auf den Bus. Als Scheduling Variante können verschiedene Verfahren angewendet werden, mehr dazu in [2].
Des Weiteren steuert die CU die SH-Modes, in dem sie die Verarbeitung der Daten durch den SH-Mode mittels den entsprechenden Steuersignalen aktiviert und deaktiviert.
Durch ein *Scorebord* wird in der CU die Kausalität der zu verarbeitenden Daten während der Ausführung des *Eingabe-Verarbeitung-Ausgabe* (EVA) Zyklus sicher gestellt [3].

Die Kommunikation zwischen den einzelnen Komponenten des IFB erfolgt über ein *fully interlocked protocol*. Der Ablauf einer minimalen Beispielkommunikation über den IFB wird im folgenden Kapitel verdeutlicht.

Datenfluss im IFB

Um die Funktionsweise und die Ablaufstruktur einer Kommunikation über den IFB zu verdeutlichen, wird hier anhand eines Beispiels von zwei inkompatiblen Tasks – Task A und Task B – der Datenaustausch illustriert (siehe Abbildung 2.4).
Da die Tasks nicht direkt miteinander kommunizieren können, wird ein IFB eingesetzt, über den die beiden Kommunikationspartner (transparent) ihre Daten austauschen.

Schritt 1 Der PH-Mode A bewirbt sich bei der CU um den Zugriff auf den Datenbus zum Data Reader. Nachdem der Scheduler der CU den Bus zuteilt, werden die Daten vom Task A gelesen.

Schritt 2 Der PH-Mode A leitet die gelesenen Nutzdaten zum Data Reader (Zwischenspeicher).

Schritt 3 Das Scoreboard startet den für die Datenverarbeitung entsprechenden SH-Mode, der die Daten aus dem Data Reader liest und modifiziert.

Schritt 4 Die modifizierten Daten werden von dem SH-Mode in den Data Writer geschrieben.

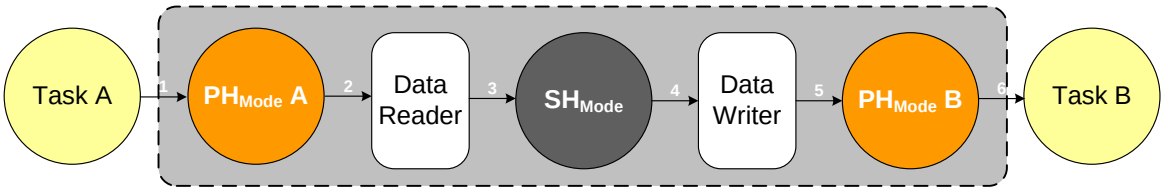

Abbildung 2.4: IFB Datenfluss

Schritt 5 Ist der Datenbus zum Data Writer von keinem anderen SH-Mode mehr belegt (SH-Mode ist fertig), so kann sich der PH-Mode B um den Zugriff darauf bewerben und danach die Daten aus dem Data Writer in das ausgehende Protokoll integrieren [3].

Schritt 6 Darauf sendet der PH-Mode B die Daten an den Task B.

2.3 Protokolle

Um die Abbildung der Protokolldaten erklären zu können, wird an dieser Stelle genauer auf den in vorangegangenen Abschnitten oft benutzten Begriff des *Protokolls* eingegangen.
Kommunikationsprotokolle definieren Regeln, welche das Format, den Inhalt, die Bedeutung und die Reihenfolge der gesendeten Nachrichten zwischen den Kommunikationspartnern festlegen. Anders ausgedrückt, Protokolle definieren die Kommunikation zwischen verschiedenen Kommunikationsteilnehmern. Dabei kann die Kommunikation direkt zwischen den Beteiligten oder über einen Bus stattfinden [8]. Der Inhalt der gesendeten Informationen wird in die eigentlichen Daten (*Nutzdaten*) und Signale zur Steuerung des Ablaufs (*protocol control information*) unterschieden.

Protokoll einer IFD

Wie bereits in Kapitel 2.1 erwähnt, sind Protokolle im *Interface Synthesis Design Flow* als Teil einer Schnittstellenbeschreibung definiert. Sie beschreiben eine Menge von Protokollpins (*PP*) und das Verhalten derer Signale in Form eines endlichen Zustandsautomaten.
Die Protokollpins repräsentieren gerichtete virtuelle Verbindungen und werden eingesetzt, um Protokolle unabhängig von den physikalischen Schnittstellen beschreiben zu können. Die Abbildung von PPs auf die physikalischen Pins einer Schnittstelle wird in der sogenannten ProtocolMap definiert. Die Verhaltensbeschreibung der Signale wird durch die Zustandsfolgen, und die Signalausgaben durch die entsprechende Zustands-

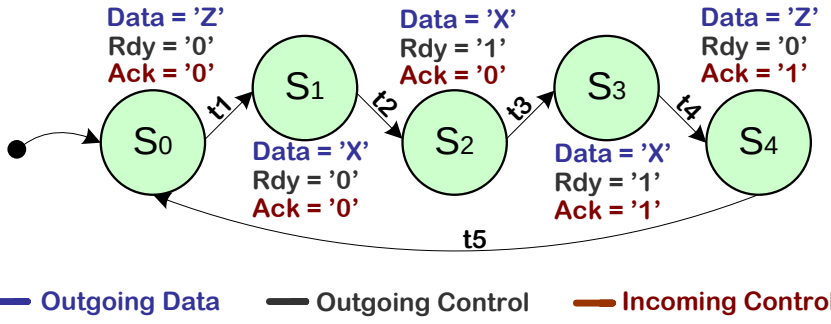

Abbildung 2.5: Protokoll-Beispiel einer Schnittstellenbeschreibung

ausgabe der FSM modelliert.
Um Protokolle zu beschreiben wurde daher die erweiterte Definition von endlichen Zustandsautomaten gewählt:

Definition 2.1 *Protokoll einer Schnittstellenbeschreibung*

$Protokoll = (S,\ TC,\ PP,\ \delta,\ \lambda,\ s_0)$

S	:	Endliche Menge an Zuständen
TC	:	Endliche Menge an Zustandsübergangsbedingungen
PP	:	Endliche Menge an Protokollpins
δ	$: S \times TC \rightarrow S$	Zustandsübergangsfunktion
λ	$: S \rightarrow (PP \mapsto Val)$	Ausgabefunktion
s_0	$\in S$	Startzustand

wobei gilt

$PP = (Richtung \times Datentyp \times Eigenschaft)$

$Val = (Wert \times Richtung \times UseCase)$

mit

$Richtung \in \{incoming,\ outgoing\}$, $Datentyp \in \{Bit, Std\_Logic, Other\}$,
$Eigenschaft \in \{control,\ data,\ control{+}data\}$,
$Wert \in \{\text{'1'},\ \text{'0'},\ \text{'X'},\ \text{'Z'},\ \ldots\}$ *und* $UseCase \in \{control,\ data\}$

Ähnlich den Zustandsautomaten, besitzt das so definierte Protokoll eine Menge von Zuständen S und den Startzustand $s_0 \in S$. Die Zustandsübergangsbedingungen TC des Protokolls werden als Zeiten bzw. Zeitintervalle modelliert. Das Pendant zum Ausgabealphabet sind die PP (Protokollpins) als Kreuzprodukt aus *Richtung*, *Datentyp* und *Eigenschaft* definiert. Die Zustandsübergangsfunktion δ weist jedem Zustand in Abhängigkeit von der TC einen neuen Zustand zu. Die Ausgabefunktion λ ordnet im jeweiligen Zustand, jedem Protokollpin einen Signalwert Val zu, der wiederum das Kreuzprodukt aus *Wert*, *Richtung* und *UseCase* darstellt. Somit ergeben sich für dynamische Signalwerte (Val) folgende mögliche Anordnungen, wobei $\bullet$ für einen beliebigen Wert steht:

($\bullet$, *incoming*, *control*)
($\bullet$, *incoming*, *data*)
($\bullet$, *outgoing*, *control*)
($\bullet$, *outgoing*, *data*)

In der Abbildung 2.5 ist beispielsweise ein Protokoll nach Definition 2.1 grafisch dargestellt. Das Protokoll beschreibt auf *fully interlocked handshake* basierende Kommunikation zwischen dem Sender und Empfänger. Die Signalwertzuweisungen an die Protokollpins *Data*, *Rdy* und *Ack* findet in den Zuständen s_0 - s_4 statt, wobei s_0 der Startzustand ist. Die Zustandsübergangsbedingungen t_1 - t_5 sind als gerichtete Kanten, die jeweils zwei Zustände miteinander verbinden, modelliert.

Es ist die Aufgabe des Systemdesigners im Rahmen der Schnittstellenbeschreibungen für die eingesetzten Kommunikationskomponenten (z.B. IPs) auch die jeweiligen Protokolle

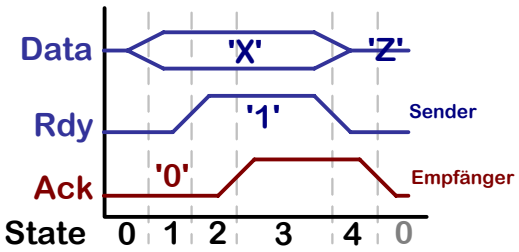

Abbildung 2.6: Visualisierung zu Protokoll in Abbildung 2.5

bereit zu stellen[2]. Diese Protokolle werden in der automatischen Schnittstellensynthese für die Generierung der PH-Modes im IFB eingesetzt. Das Java Tool "IFS-Editor" bietet dem Entwickler neben der Möglichkeit die existierende Protokolle zu importieren und anzupassen, auch eigene zu modellieren. Der Modellierungsprozess wird dabei durch die Fähigkeit des "IFS-Editor" die Protokolle blockweise zu visualisieren unterstützt. Hierfür wird das Protokoll in einzelne Mengen nicht verzweigender Zustände (*basic block, BB*) unterteilt (siehe "Protokoll-Grundblöcke" auf Seite 15), die dann einzeln als Flankendiagramm dargestellt werden können. Die Abbildung 2.6 stellt beispielsweise das Protokoll aus dem vorangegangenen Beispiel (siehe Abbildung 2.5) als Flankendiagramm dar.

ProtocolFSM

Während der Synthese des IFB-Models dienen die Protokolle aus den Schnittstellenbeschreibungen als Grundlage für die Erzeugung der PH-Modes (*Stubs*) des Protocol Handlers im IFB. Die synthetisierte Stubs übernehmen jeweils die Kommunikation mit den externen Tasks und Medien indem sie deren komplementäres Protokoll ausführen. Sie ermöglichen somit den Datentransfer zwischen den über IFD verbundenen Tasks und Medien (siehe Abbildung 2.3 auf Seite 9). Für jede Kommunikationsverbindung mit einer angebundenen Task oder einem Medium wird im PH ein Stub erzeugt.

Das Verhalten der Stubs wird ebenfalls durch einen endlichen Zustandsautomaten, der *ProtocolFSM*, beschrieben. Zur automatischen Generierung der *ProtocolFSM* bzw. eines Stubs, wird das Protokoll aus der IFD der Task extrahiert. Anhand der Wertänderungen auf den Daten- und Kontrollleitungen des Protokolls werden die Zustände mit ihren Ausgaben und Zustandsübergangsbedingungen der *ProtocolFSM* erzeugt.
Um die Erzeugung einer *ProtocolFSM* zu verdeutlichen, wird nachfolgend die Generierung eines sendenden Stubs aus dem Protokoll in der Abbildung 2.5 schrittweise erläutert.

Die Wertänderung auf den ausgehenden (*outgoing*) Daten- bzw. Kontrollleitungen des Protokolls führen zur Erzeugung eines Zustandes in der *ProtocolFSM*. Die Änderungen auf den eingehenden Kontrollleitungen (*incoming control*) werden als Zustandsübergangbedingungen übernommen.
Weil sich nach dem Zustandswechsel von s_0 auf s_1 (siehe Abbildung 2.5) der Wert der

[2]Bereits existierende Schnittstellenbeschreibungen lassen sich im IFS-Flow wiederverwenden.

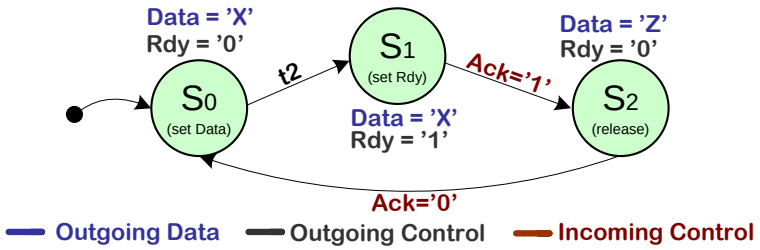

Abbildung 2.7: ProtocolFSM

Datenleitung *Data* von 'Z' auf 'X' ändert, wird der Zustand s_0 der *ProtocolFSM* (siehe Abbildung 2.7) durch den Synthesealgorithmus generiert. Die Ausgabe des Protokollzustandes s_1 wird auch zur Ausgabe des neu entstandenen *ProtocolFSM*-Zustandes s_0. Der Zustand s_1 der *ProtocolFSM* entsteht durch die Wertänderung von '0' auf '1' auf der Kontrollleitung *Rdy* beim Wechsel von s_1 auf s_2 (siehe Abbildung 2.5). Die Zustandsübergangsbedingung *'Ack=1'* in den Zustand s_2 der *ProtocolFSM* – sowie der Zustand selber – resultieren jeweils aus den Wertänderungen auf den Protokollpins *Ack* (*incoming control*) und *Data* (*outgoing data*). Im letzten Schritt wird die Änderung der Protokoll-Ausgabe auf dem PP 'Ack' beim Zustandswechsel von s_4 auf s_0 als Zustandsübergangbedingung *'Ack=0'* in den Zustand s_0 der *ProtocolFSM* übernommen.

Bis auf zwei Einschränkungen bzw. Erweiterungen wird die ProtocolFSM formal ebenfalls nach Definition 2.1 beschrieben:

- Die Kombination (•, *incoming, control*) für die mögliche Signalwerte (*Val*) in der Ausgabefunktion λ der *ProtocolFSM* ist nicht definiert.
- Die Zustandsübergänge der *ProtocolFSM* sind nun nicht mehr nur von Zeiten abhängig. Das heißt die TCs der *ProtocolFSM* werden nicht nur als Zeitintervalle modelliert, sondern auch als Zustandsübergangsbedingungen, die auf eingehende Steuersignale der angeschlossenen Kommunikationskomponente reagieren.

Der Stub ist somit in der Lage zwischen Nutzdaten und Steuerdaten zu unterscheiden. Die Steuerdaten werden als redundanter Teil des Protokolls von der ProtocolFSM generiert bzw. konsumiert. Ausschließlich die Nutzdaten werden zur Weiterverarbeitung an den SH übertragen [2]. Für die automatische Transformation der Daten im Schnittstellenadapter werden zusätzliche Informationen benötigt, die in Form des IFD-Mappings definiert werden. Neben der in den Protokollen festgelegten *Syntax* wird durch das IFD-Mapping (siehe Kapitel 3) die *Semantik* der Daten in Form von Abbildungsvorschriften festgelegt. Erst durch die einzelnen Abbildungsvorschriften im IFD-Mapping ist es dem Synthesealgorithmus möglich die automatische Datenverarbeitung im IFB zu generieren. Grundlage für das IFD-Mapping sind die *Datenpakete*, die ihrerseits aus den *Grundblöcken* abgeleitet werden.

Protokoll-Grundblöcke

Ein Protokoll besteht aus einer Menge miteinander über gerichtete Kanten verbundenen Zustände. Die Zustände lassen sich in disjunkte Mengen unterteilen, die sogenannten *Grundblöcke* (*basic block, BB*). Die Unterteilung der Zustände in *Grundblöcke* basiert auf der Tatsache, dass in einem syntaktisch korrekten Protokoll die Übertragung der Datenpakete immer vollständig und ohne Unterbrechungen ablaufen muss. Eine geordnete und kreisfreie Zustandsfolge mit höchstens einer Verzweigung – und zwar nach dem letzten Zustand – stellt so eine gültige Menge dar. Dabei ist der erste Zustand der Folge entweder der Startzustand oder der Nachfolger von mehreren Zuständen außerhalb der Folge. Diese Zustandsfolgen repräsentieren für die Datenübertragung zuständige Abschnitte des Protokolls. Jede Verzweigung oder Zusammenführung in der Zustandsfolge dient der Steuerung des Protokollablaufs.
In der Abbildung 2.8 ist ein Protokoll mit fünf Zuständen s_0 bis s_4 und deren Aufteilung in einzelne *Grundblöcke* BB_1 bis BB_4 dargestellt. Der Startzustand s_0 bildet immer den Anfang eines *Grundblocks* und wird daher in die Menge der Zustände von Grundblock BB_1 aufgenommen. Da die Zustände s_1 bzw. s_3 jeweils die Folgezustände von s_0 sind (Verzweigung), bilden sie den Anfang der neuen *Grundblöcke* BB_2 und BB_3. Der Zustand s_2 besitzt nur einen Vorgängerzustand s_1, und gehört daher in denselben *Grundblock* BB_2. Den neuen und letzten *Grundblock* BB_4 bildet der Zustand s_4 als Nachfolger von mehreren (s_2, s_3) Zuständen (Zusammenführung).

Während der Schnittstellensynthese werden die ProtocolFSMs der über den IFB verbundenen Kommunikationskomponenten in *Grundblöcke* unterteilt. Die Zerlegung der Protokolle in ihre *Grundblöcke* ist an mehreren Stellen im IFS-Flow relevant [8]. Unter anderem werden die *Grundblöcke* für die Visualisierung der Protokolle als Flankendiagramm genutzt, da sie per Definition eine Menge nicht verzweigender Zustände bilden. Die Abbildungsvorschriften im IFD-Mapping werden auf sogenannten Datenpaketen (*data package, P*) definiert, die wiederum aus den einzelnen *Grundblöcken* der ProtocolFSM abgeleitet werden.

Datenpakete

Ein *Datenpaket* (*data package*) ist ein abstrakter Datentyp innerhalb des IFS-Flows und stellt die zentrale Datenstruktur für das IFD-Mapping dar. Die Abbildungsvorschriften für die Synthese der Schnittstellenadapter werden auf den *Datenpaketen* definiert. Ab-

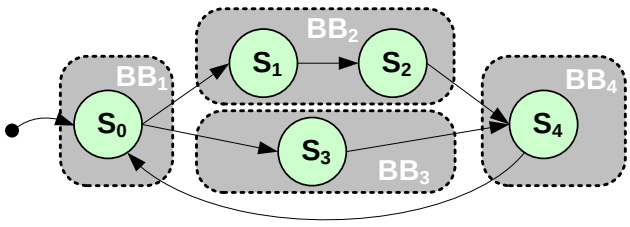

Abbildung 2.8: Grundblöcke eines Protokolls

geleitet werden die *Datenpakete* aus den Grundblöcken eines Protokolls. Genauer gesagt stellen sie eine Abstraktion von den Grundblöcken dar um es dem Systemdesigner zu ermöglichen die Datenabbildungen auf eine einfache und intuitive Weise zu definieren. Die Nutzung der Grundblöcke als Ausgangstruktur für die Datenabbildungen innerhalb von IFD-Mapping ist die Konsequenz aus den Eigenschaften der Protokollgrundblöcke. Denn wie bereits erklärt sind es die Grundblöcke die eine vollständige und unterbrechungsfreie Datenpaketübertragung innerhalb des Protokolls realisieren. Es wird für den IFS-Flow sogar vorausgesetzt, dass die *Datenpakete* nicht über die Grenzen eines Grundblocks hinausgehen [8].

Für die *Datenpaketdefinition* werden die Protokollgrundblöcke zunächst als Matrizen nachgebildet. Die Spalten einer Matrix stellen die Zustände des Grundblocks und Zeilen die Protokollpins dar (siehe Abbildung 2.9). In den einzelnen Zellen der Matrix sind die Signalwerte (*Val*) eines Protokolls als Datenbits (*D*) oder Kontrollbits (*C*) mit der jeweiligen Richtung (*outgoing O, incoming I*) als Einträge vorhanden. Daraufhin werden die Protokoll-Matrizen auf *Datenpakete* untersucht.

Datenpakete sind in der IFB-Terminologie zusammenhängende Datenbits (Nutzdaten) die von einem Protokoll übertragen werden. Ermittelt werden die *Datenpakete* durch die zeilenweise Untersuchung der Matrizen nach zusammenhängenden Datenbits. Hierfür werden die benachbarten Zellen mit gleichen Richtungen (*incoming, outgoing*) zu Teilpaketen zusammengefasst. Anschließend werden aus Teilpaketen gleicher Länge und gleicher Anfangsposition auf der X-Achse der Matrix (Zustände) die Datenpaketen gebildet. In der Abbildung 2.10 sind die *Datenpakete* 1 bis 6 zur der Protokoll-Matrix aus der Abbildung 2.9 farblich hervorgehoben.

Während der automatischen Generierung des Schnittstellenadapters fließen die vom Designer der Systemarchitektur definierte Datenabbildungen in den Generierungsprozess mit ein. Durch die Abbildungsvorschriften können jedoch Abhängigkeiten zwischen den einzelnen Datenpaketen entstehen, die durch die “Gruppenbildung” der Datenpakete

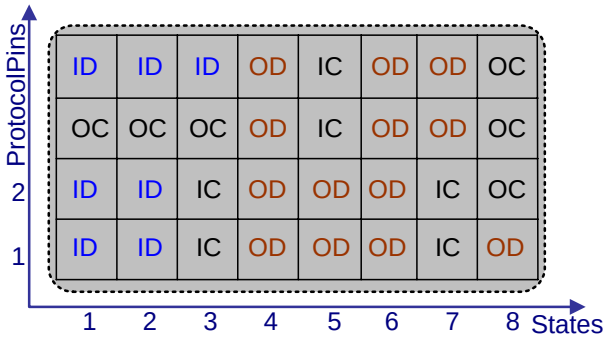

Abbildung 2.9: Grundblock als Matrix

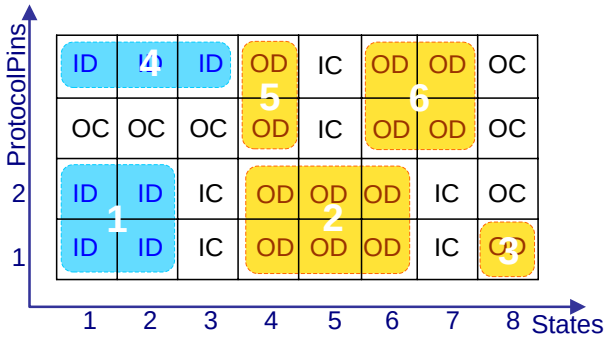

Abbildung 2.10: Datenpakete

(*Frames*) aufgelöst werden müssen.

Frames

Frames sind Mengen aus sich überlappenden Datenpaketen die ohne Unterbrechung zwischen PH und SH übertragen werden. Wie im vorangegangenen Beispiel (siehe Abbildung 2.10) zu sehen, überschneiden sich die Datenpakete 1 und 4 in den Zuständen 1 bis 2. Während der Ausführung der ProtocolFSM werden in den beiden Zuständen demzufolge die Daten aus beiden Datenpaketen übertragen. Im Zustand 2 ist die Übertragung des Datenpaketes 1 vollständig abgeschlossen, jedoch darf die Kommunikation an dieser Stelle nicht unterbrochen werden. Denn um das Datenpaket 4 seinerseits auch zuende zu übertragen, muss zusätzlich der Zustand 3 "ausgeführt" werden. Um eine unterbrechungsfreie Ausführung des Protokolls zu gewährleisten, werden daher die beiden Datenpakete IFB intern zu einem *Frame A* zusammengefasst. Mit der Übertragung der Datenpakete darf dann erst begonnen werden, wenn alle Datenpakete des *Frames* auch vollständig übertragen werden können. Demnach stellen die Frames die zwischen dem PH und SH übertragenden Einheiten dar.

Zusätzlich zu den Datenpaketen 1 und 4 existieren in dem Beispiel weitere, sich überlappende, Datenpakete. So bilden die Datenpakete 2, 5 und 6 den nächsten *Frame B* (siehe Abbildung 2.11). Der *Frame C* enthält nur ein Datenpaket 3, da dieser keine Überschneidungen mit anderen Paketen hat. Folglich enthält jeder Frame mindestens ein Datenpaket.

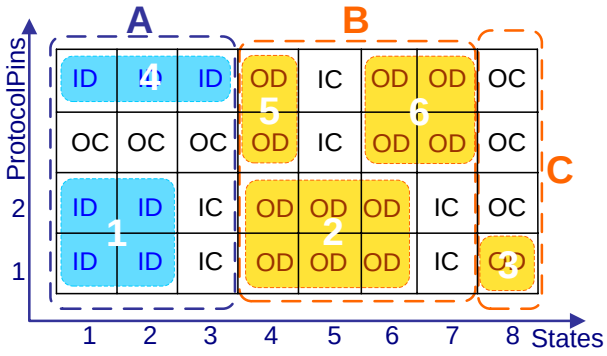

Abbildung 2.11: Frames eines Grundblocks

Wie bereits erläutert basiert die IFB-interne Übertragung der Datenpakete zwischen den Tasks auf semantikfreien Protokollen. Das Protokoll in einer IFD legt nur die Syntax der Daten fest, ohne etwas darüber auszusagen, was die Datenpakete bedeuten oder was letztendlich damit passieren soll. Wie die Datenpakete aus dem Senderprotokoll auf das Empfängerprotokoll abgebildet werden sollen, wird durch das IFD-Mapping festgelegt. Das IFD-Mapping ergänzt somit die Protokollbeschreibungen um die fehlende Semantik. Die formalen Grundlagen für das in Kapitel 3 vorgestellte Konzept des IFD-Mapping liefert das folgende Kapitel 2.4.

2.4 Grammatiken und formale Sprachen

In einer natürlichen Sprache (z.B. Deutsch, Englisch, usw.) dienen die Grammatiken dazu festzulegen, welche Wörter und Zeichen sowie die Kombination und Aneinanderreihung von diesen, korrekte Sätze sind. Eine Grammatik macht dabei keine Aussage über die Bedeutung von Sätzen (Semantik), sondern beschreibt lediglich die Syntax einer Sprache. So ist zum Beispiel dem Satz "Futter trocknen schnelles Auto" nur schwer eine Bedeutung zuzuordnen, er ist jedoch ein durchaus korrekter Satz der deutschen Sprache. Wenn wir die Wörter einer natürlichen Sprache als Kombination von Alphabetzeichen ansehen, dann bilden die Sätze einer natürlichen Sprache eine *formale Sprache* über dem Alphabet der natürlichen Wörter. Allerdings entzieht sich die natürliche Sprache einer vollständigen Definition, die letztendlich festlegt, welche Sätze zu der natürlichen Sprache gehören und welche nicht.

Definition 2.2 *Formale Sprache*
Sei Σ ein beliebiges Alphabet. Eine *formale Sprache* (über Σ) ist jede beliebige Teilmenge von Σ^*. Wobei Σ^* eine Menge von Wörtern ist, die aus der Konkatenation von Symbolen oder Zeichen aus dem – in der Regel endlichen – Alphabet Σ entstehen [10].

Sei z.B. $\Sigma = \{(,\ ),\ +,\ \text{-},\ *,\ /,\ a\}$ das endliche Alphabet. So kann man eine Sprache der korrekt geklammerten arithmetischen Ausdrücke $EXPR \subseteq \Sigma^*$ so definieren, wobei a als Platzhalter für beliebige Konstanten oder Variablen dienen soll:

$$\begin{aligned} (a\ \text{-}\ a) * a + a\ /\ (a + a)\ \text{-}\ a &\in EXPR \\ (((a))) &\in EXPR \\ ((a)\ \text{-}\ a(&\notin EXPR \end{aligned}$$

Die meisten Programmiersprachen, wie z.B. Java oder C, sind formale Sprachen. So bilden Wörter in Java die jeweiligen Programme. Das Alphabet von Java sind die Schlüsselwörter und Zeichen, welche in der Java Sprachdefinition festgelegt sind.

Um solche Sprachen formal definieren zu können, benötigt man eine Beschreibungsmöglichkeit für Sprachen. Eine Möglichkeit der Beschreibung bieten Automaten, eine andere die bereits erwähnten Grammatiken, wobei wir uns im folgenden primär auf Grammatiken konzentrieren. Für weitere Details kann in [12] nachgeschlagen werden.

Ein (formaler) Text besteht aus Symbolen, die auch *Terminalsymbole* genannt werden. In der einschlägigen Literatur zum Thema Grammatiken werden dafür üblicherweise kleingeschriebene Symbole verwendet (a,b,c, ...). In den Programmiersprachen kommen auch andere Symbole, wie Satzzeichen und Schlüsselwörter IF, $ELSE$, $WHILE$, ..., vor. Ein Text besteht demnach aus kleingeschriebenen *Terminal-Symbolen* wie z.B. *aaabcbbba*. Desweiteren werden in formalen Grammatiken *Nichtterminalen* als Pseudosymbole bzw. Platzhalter, die im Text nicht auftauchen, verwendet. Üblicherweise werden dafür großgeschriebene Symbole verwendet. Grammatiken werden in Form von

Produktionsregeln definiert, in denen die *Nichtrerminale* durch *Terminale* oder Kombination aus *Terminalen* und *Nichtterminalen* ersetzt werden. Eine Beispiel-Grammatik könnte z.B. so aussehen:

Beispiel 2.1

$$\begin{array}{lcl} A & \longrightarrow & Ba \\ B & \longrightarrow & b \\ B & \longrightarrow & Cb \\ C & \longrightarrow & Bcc \\ C & \longrightarrow & Cc \end{array}$$

Eine der *Nichtterminalen* (*A, B, oder C*) wird als *Startsymbol* definiert, hier z.B. *A*. Aufgrund der *Produktionsregeln* können nun gültige Texte produziert werden.

$A \rightarrow Ba \rightarrow ba$
oder
$A \rightarrow Ba \rightarrow Cba \rightarrow Bccba \rightarrow bccba$
oder
$A \rightarrow Ba \rightarrow Cba \rightarrow Bccba \rightarrow Cbccba \rightarrow Bccbccba \rightarrow \ldots \rightarrow bccbccbccbccba$
oder
$A \rightarrow Ba \rightarrow Cba \rightarrow Ccba \rightarrow Cccba \rightarrow Ccccba \rightarrow \ldots \rightarrow bccccccccccba$

Der obige Text *bccbccbccbccba* kann also durch die sukzessive Anwendung der Regeln aus dem Startsymbol *A* produziert werden. Daher auch die Bezeichnung Produktionsregeln. Die so produzierte Texte sind gültig, im Sinne der Grammatik.

Definition 2.3 *Grammatik*
Eine Grammatik ist ein *4-Tuppel* $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt:
V ist eine endliche Menge, die Menge der *Variablen* oder *Nichtterminalen*. Σ ist eine endliche Menge, das *Terminalalphabet*. Es muss gelten: $V \cap \Sigma = \varnothing$. P ist die endliche Menge der *Regeln oder Produktionen*. Formal ist P eine endliche Teilmenge von $(V \cup \Sigma)^+ \times (V \cup \Sigma)^*$. $S \in V$ ist die *Startvariable*. Seien $u, v \in (V \cup \Sigma)^*$. Wir definieren die Relation $u \Rightarrow_G v$ (in Worten: u geht unter G unmittelbar über in v, kurz $u \Rightarrow v$, falls G klar ist), falls u und v die Form haben:

$$\begin{array}{c} u = xyz \\ v = xy'z \text{ mit } x, z \in (V \cup \Sigma)^* \\ \text{und } y \rightarrow y' \text{ eine Regel in } P \text{ ist.} \end{array}$$

Die von G dargestellte (erzeugte, definierte) Sprache ist

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Man klassifiziert verschiedene Typen von Grammatiken in der so genannten Chomsky-Hierarchie, genannt nach dem amerikanischen Sprachwissenschaftler Noam Chomsky (geb. 1928).

Definition 2.4 *Chomsky-Hierarchie*
Es sei $G = (V, \Sigma, P, S)$ eine Grammatik.

- Jede solche Grammatik heißt auch Grammatik vom *Typ 0*.
- Eine Grammatik ist vom *Typ 1* oder *kontextsensitiv*, falls für alle Regeln $w_1 \rightarrow w_2$ gilt: $|w_1| \leq |w_2|$ (Länge von w_1 ist kleiner oder gleich der Länge von w_2).
- Eine *Typ 1 - Grammatik* ist vom *Typ 2* oder *kontextfrei*, falls für alle Regeln $w_1 \rightarrow w_2$ in P gilt, dass w_1 eine einzelne Variable (*Nichtterminale*) ist, d.h. $w_1 \in V$.
- Eine *Typ 2 - Grammatik* ist vom *Typ 3* oder *regulär*, falls zusätzlich gilt: $w_2 \in \Sigma \cup \Sigma V$, d.h., die rechten Seiten von Regeln sind entweder einzelne *Terminalzeichen* oder ein *Terminalzeichen* gefolgt von einer *Variablen*.

Eine Sprache $L \subseteq \Sigma^*$ heißt von *Typ 0* (*Typ 1, Typ 2, Typ3*), falls es eine *Typ 0* (*Typ 1, Typ2, Typ3*)*-Grammatik* G gibt mit $L(G) = L$ [10].

In den Kapitel 3 und 4 dieser Arbeit findet die *kontextfreie Grammatik (Typ 2)* und die von ihr beschriebene *kontextfreie Sprache* Anwendung.

Von John Backus entwickelt und nach ihm benannt, später in *Backus-Naur Form* umbenannt, ist *BNF* eine kompakte formale Metasprache, die benutzt wird um *kontextfreie Grammatiken* einfach und lesbar darzustellen. Die *BNF* definiert, wie in der vorherigen Definition 2.3, *Produktionsregeln*, *Nichtterminalen (Variablen)* und *Terminalen*. Dabei dient ein "|" als Alternative und die Zeichenfolge "::=" wird zur Definition von Produktionsregeln verwendet. Zusätzlich werden die *Nichtterminalen* mit spitzen Klammern "<" und ">" umschlossen. Die Grammatik aus dem Beispiel 2.4 wird in *BNF* folgendermaßen definiert:

$$
\begin{aligned}
&< A > ::= < B > a \\
&< B > ::= b \mid < C > b \\
&< C > ::= < B > cc \mid < C > c
\end{aligned}
$$

Die ***e**rweiterte* Backus-Naur-Form ***E**BNF* ist eine Erweiterung der *BNF* und wurde von der ISO standardisiert. Durch Hinzufügen folgender Elemente wird aus *BNF* eine *EBNF*:

- Gruppierung durch Klammern
- Optionalität
- Wiederholungen
- Mehrzeilige Ausdrücke möglich, da jeder Ausdruck mit ";" beendet werden muss.

Außerdem werden die *Nichtterminalen* nicht in die spitze Klammern gesetzt. Anstelle dessen werden *Terminale* von einfachen oder doppelten Anführungszeichen umschlossen. Damit lassen sich die Symbole “<”, “>”, und “::=” besser von der Notationsart der *BNF* unterscheiden, falls sie in der Definition der Grammatik verwendet werden. Die Zeichen für Optionalität und Wiederholungen haben folgende Bedeutung:

- [. . .] - eins oder keins
- {. . . } - beliebig viele, auch keins
- + - beliebig viele, mindestens jedoch eins
- – - mit Ausnahme von

Wieder das Beispiel 2.4, diesmal in *EBNF*:

$$
\begin{array}{lcl}
A & = & \text{B “a”;} \\
B & = & \text{[C] “b”;} \\
C & = & B \text{ “}cc\text{”\{“}c\text{”\};}
\end{array}
$$

2.5 Compiler und Parser

Ein Compiler (Übersetzer) ist ein Computerprogramm, das ein in einer Quellsprache geschriebenes Programm in ein semantisch äquivalentes Programm einer Zielsprache übersetzt [13]. Der Übersetzungsvorgang wird als *Kompilieren* bezeichnet. Generell lässt sich der Kompiliervorgang in zwei Phasen einteilen (siehe Abbildung 2.12):

- Die *Analysephase* (auch “Fronted”), die den Quelltext analysiert.
- Die *Synthesephase* (auch “Backend”), die das Zielprogramm erzeugt.

In der Analysephase wird zuerst die *lexikalische* Analyse durchgeführt. Dabei wird der eingelesene Quelltext in so genannten Token verschiedener Klassen eingeteilt. Die Token (Folge von Zeichen) sind lexikalische Grundeinheiten, die im einfachsten Fall einzelne Buchstaben aus dem Text sein können. Die Klassen könnten z.B. Bezeichner, Zahlen, Operatoren und Schlüsselwörter sein (siehe Abbildung 2.13).

Nach der *lexikalischen* Analyse folgt die *syntaktische* Analyse (von einem *Parser* durchgeführt), in der überprüft wird, ob der eingelesene Quellcode (Text) formal richtig ist. Das heißt, es wird überprüft, ob die Syntax (Grammatik) der Quellsprache entspricht. Dabei erstellt der Parser aus der Eingabe einen Syntaxbaum (siehe Abbildung 2.14).

Es folgt nun die *semantische* Analyse, die die statische Semantik überprüft, sprich die “logische Rahmenbedingungen”. Zum Beispiel muss eine Variable deklariert worden sein,

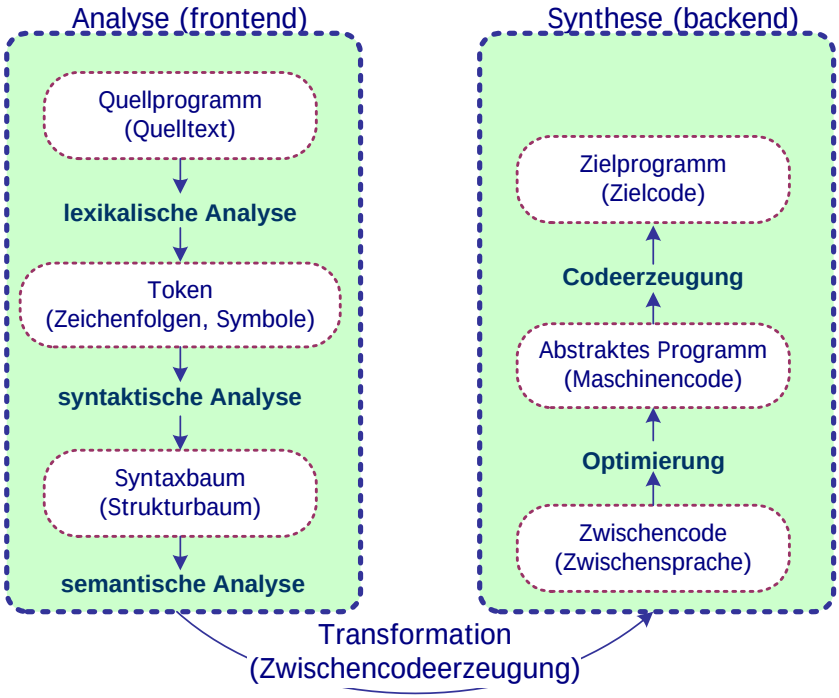

Abbildung 2.12: Übersetzungsstruktur

bevor sie verwendet wird, und Zuweisungen müssen mit kompatiblen Datentypen erfolgen. Dabei werden die Knoten des vom Parser generierten Ableitungsbaums, mit Attributen versehen, welche die benötigten Informationen enthalten. So kann zum Beispiel eine Liste aller deklarierten Variablen erstellt werden.

Eine sich anschließende *Synthesephase* generiert aus dem in der *Analysephase* erstellten Baum den Programmcode der Zielsprache.

JavaCC

JavaCC (Java Compiler Compiler) ist ein in Java implementierter Generator, der aus einer vordefinierten Grammatik einen lexikalischen Analysierer und Parser in Javacode erzeugt. Entwickelt wurde JavaCC ursprünglich, um die Programmiersprachenimplementierung – daher auch Compiler Compiler – zu vereinfachen, es kann jedoch nicht nur zur Entwicklung von Programmiersprachen eingesetzt werden.

Die Grammatik für den Generierungsprozess von JavaCC wird in EBNF-ähnlichen Notation spezifiziert. Die so definierte Produktionsregeln der Grammatik werden dabei in einer .jj-Datei gespeichert und, falls fehlerfrei, von JavaCC dann für die automatische Generierung eines lexikalischen Analysierers (dem so genannten *Token Manager*, siehe Abbildung 2.13) und eines Parsers verwendet (siehe Abbildung 2.14). Dabei entstandenen Javaklassen können anschließend von dem Entwickler entweder direkt genutzt oder auch für weitere Anforderungen angepasst werden.

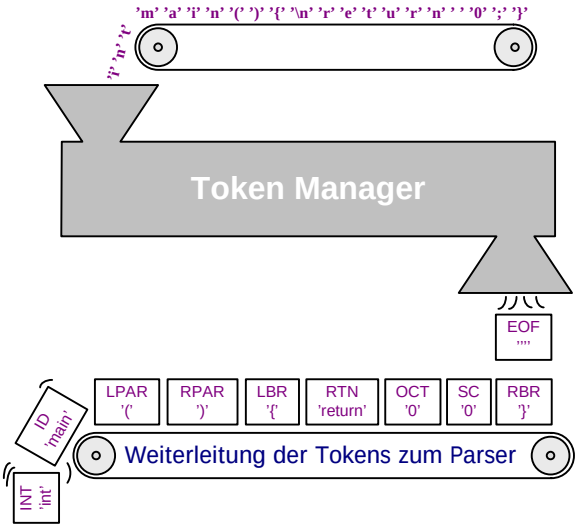

Abbildung 2.13: Tokenmanager [7]

Von Haus aus kann JavaCC keine Ableitungsbäume für die Nutzung in der semantischen Analyse generieren. Es existieren jedoch zumindest zwei frei verfügbare Tools wie *JJTree* und *JTB* [7], die es durch eine Integration in den Generierungsprozess ermöglichen, ohne großen Aufwand solche Bäume zu erstellen. Auch kann JavaCC nicht automatisch den Programmcode der Zielsprache generieren, was jedoch von dem Entwickler kompensiert werden kann, indem er die Informationen aus dem Ableitungsbaum an den Algorithmus für die Codeerzeugung weitergibt.

Weitere Informationen über JavaCC stehen unter [7] bereit, wo der Leser weiterführende Dokumentation sowie freie Downloads vorfinden kann.

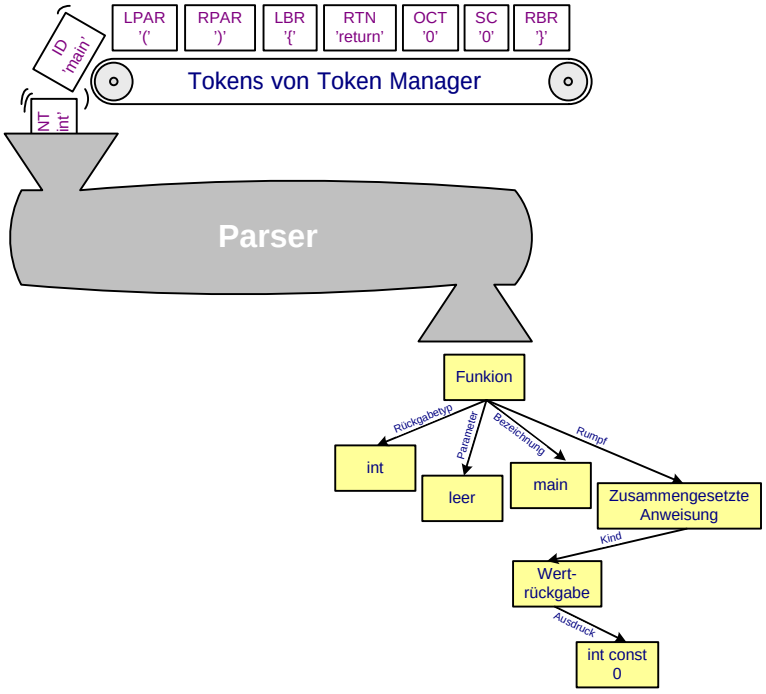

Abbildung 2.14: Parser

3 IFD-Mapping

Wie bereits im vorangegangenen Kapitel beschrieben, wurde der Ansatz des IFS-Flows entwickelt, um Schnittstellenadapter für heterogene Kommunikationskomponenten automatisch zu generieren. Besonderes Augenmerk lag dabei auf der Integration und flexiblen Wiederverwendung von Komponenten im IP basierten Hardwareentwurf. Die Optimierung der Entwicklungszeiten von Prototypen durch die automatische Adaptersynthese war ebenso ein wichtiger Aspekt.
Die Verarbeitung der Daten im Schnittstellenadapter verläuft, gesteuert von der CU, stufenweise nach dem Eingabe-Verarbeitung-Ausgabe (EVA) Prinzip. Im ersten Schritt werden Daten eingelesen, im zweiten werden diese verarbeitet und im dritten und letzten Schritt werden sie dann versendet [3]. Die Stubs im PH führen die Eingabe und Ausgabe Stufen aus, indem sie die Datenpakete aus dem Kommunikationskanal extrahieren und nach deren Verarbeitung wieder integrieren. Die SH-Modes des SH verarbeiten die extrahierten Datenpakete und werden durch das IFD-Mapping definiert. Sie stellen die Verarbeitungsstufe im EVA Schema des IFB dar.

Existiernde Ansätze

Es existieren bereits zahlreiche Veröffentlichungen zum Thema der automatischen Synthese von inkompatiblen Protokollen. Eine Vielzahl von IPs unterschiedlicher Hersteller bringt nicht nur Vorteile für einen Hardwaredesigner. Einerseits kann er sich aus einem großen Sortiment an IPs bedienen, um das gewünschte System baukastenartig zusammenstellen zu können, wird jedoch andererseits zwangsläufig mit einer Vielzahl an verschiedenen inkompatiblen Schnittstellen und Protokollen konfrontiert. Bis heute existierende Lösungsvorschläge können jedoch nur teilweise eine Abhilfe schaffen. So definieren Akella und McMillan [1] die Protokolle als zwei Zustandsautomaten, wobei der dritte Automat die Datentransformation repräsentiert und vom Systemdesigner als Verhaltensbeschreibung der Schnittstelle modelliert werden muss. Aus diesen FSMs wird dann ein optimierter Produktautomat gebildet, der dann die Datentransformation durchführt. Dabei müssen die zu übertragende Datenpakete die gleiche Länge aufweisen, da sie sonst nicht behandelt werden können. Ähnlich geht der Ansatz von Roberto Passerone [9] vor. Allerdings muss der Designer das Verhalten der Schnittstelle nicht mehr als FSM beschreiben und die Datenpakete können beliebig gross sein. Um das zu erreichen wird in seiner Lösung eine Beschreibung der Protokolldaten durch reguläre Ausdrücke vorausgesetzt. Dies ist jedoch problematisch, denn die Definition einer festen Protokollsemantik für die bereits existierenden und stetig neu hinzukommenden Kommunikationskomponenten ist nicht praktikabel.

Ansatz des IFD-Mapping

Im Gegensatz zu den existierenden Ansätzen wird im IFS-Flow davon ausgegangen, dass die Protokolldaten keine Semantik besitzen. Die Bedeutung der Daten entsteht erst in dem Moment, in dem die Kommunikationspartner über ihre Schnittstellen miteinander verbunden werden. Bis zu ihrer anwendungsbezogenen Verwendung werden die Daten als "Platzhalter" ohne jeglichen Semantik betrachtet. Während der Synthese muss der Systemdesigner durch die Eingabe von Abbildungsvorschriften (Regeln) den Daten die gewünschte Bedeutung zuordnen. Hierfür wurde das Konzept des *IFD-Mappings* entwickelt.

Das *Interface Definition Mapping* (IFD-Mapping) ist eine Menge von Regeln, den sogenannten *Mapping Functions*. Die einzelnen Regeln definieren die Abbildungen der eigehenden Daten auf ausgehende und haben folgende Form:

$$\textit{Mapping Function}: \textit{dataOut} \Leftarrow f_{Map}\,(\,\textit{dataIn}_1, \ldots, \textit{dataIn}_k\,)$$

Die *Mapping Function* definiert in der IFB-Terminologie die Abhängigkeit der ausgehenden Daten von eingehenden. Dabei können die eingehenden Daten auch von mehreren Kommunikationspartnern stammen. Der Begriff *"Mapping Function"* hat mit dem in der Mathematik üblichen Funktionsbegriff, wie es auf den ersten Blick erscheinen mag, nur eine bedingte Gemeinsamkeit (z.B. boolesche Funktionen). Für die Abbildung f_{Map} stehen folgenden Grundoperationen zur Verfügung:

- Zuweisung konstanter Werte
- Zuweisung eingehender Bits (ohne diese zu verändern)
- Anwendung von booleschen Funktionen auf eingehende Bits
- Verwendung eines endlichen Zustandsautomaten (FSM) zur Datenmanipulation

Die Operationen bieten die Möglichkeit konstante Werte zuzuweisen oder eingehende Bits beliebig auf ausgehende abzubilden. Auch eine Verknüpfung durch boolesche Funktionen wird unterstützt. Für die Beschreibung komplexerer Abbildungen der eingehenden Daten kann der Systemdesigner endliche Zustandsautomaten bzw. bedingte Zuweisungen definieren.

Für die Abbildungsdefinitionen der *Mapping Function* wurde im Rahmen des IFD-Mappings eine Abbildungssprache entwickelt. Diese legt die Syntax fest, in der die Abbildungen f_{Map} vom Systemdesigner definiert werden. Auf die Beschreibung der Sprache in Form einer Grammatik wird im Unterkapitel 3.4 auf der Seite 36 genauer eingegangen. Die für die Definition der *Mapping Function* zur Verfügung stehende Sprachkonstrukte werden nun anhand von Beispielen vorgestellt.

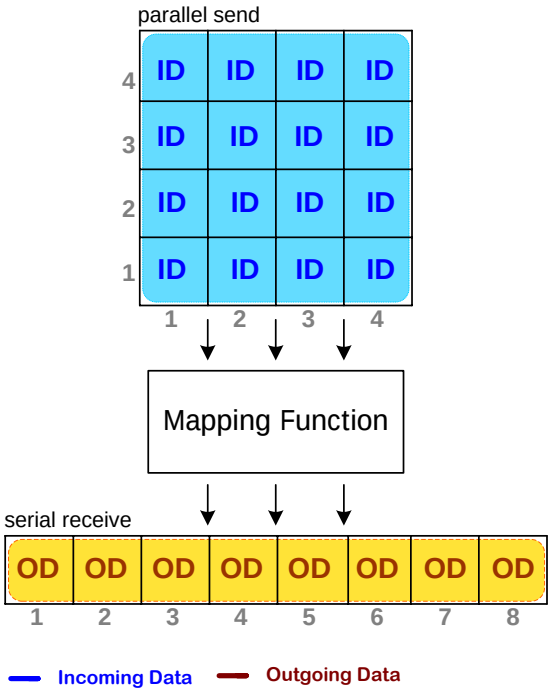

Abbildung 3.1: Anwendung der “Mapping Function”

3.1 Sprachkonstrukte

In der Abbildung 3.1 ist jeweils ein Datenpaket aus zwei verschiedenen Protokollen dargestellt. Das erste Datenpaket (oben) stammt aus einem Protokoll, das seine Daten parallel versendet. Das zweite Datenpaket wurde aus einem seriellen Protokoll generiert. Dieses Anwendungsszenario stellt genau den Fall dar, bei dem zwei Tasks mit inkompatiblen Protokollen ihre Daten über einen Schnittstellenadapter austauschen. Die Datenpakete wurden jeweils aus beiden Protokollen, wie im Unterkaptitel 2.3 auf Seite 15 beschrieben, generiert und dienen als Basis für die Definition der *Mapping Functions*. Zur Vereinfachung soll hier davon ausgegangen werden, dass jedes Datenpaket genau einen Frame darstellt.

Eine *Mapping Function* kann beispielsweise folgende Datenabbildungen beinhalten[1]:

1) $P_{out}[1{:}4] \Leftarrow P_{in}[1][1{:}4]$;
2) $P_{out}[5] \Leftarrow P_{in}[2][1]$;

[1]Die in den folgenden Beispielen genutzte Alternativschreibweise der Datenpaketbezeichnungen dient zur Vereinfachung und besseren Lesbarkeit. Sie syntaktisch korrekte schreibweise wird in Kapitel 3.4 vorgestellt. Hier reicht es anzunehmen, dass “P_{out}” für “OP_1” und P_{in} für “IP_1” steht.

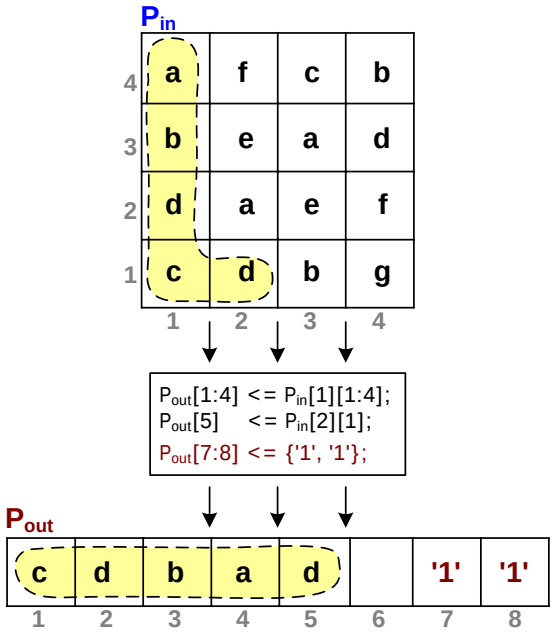

Abbildung 3.2: Zuweisung eingehender Bits und Konstanten

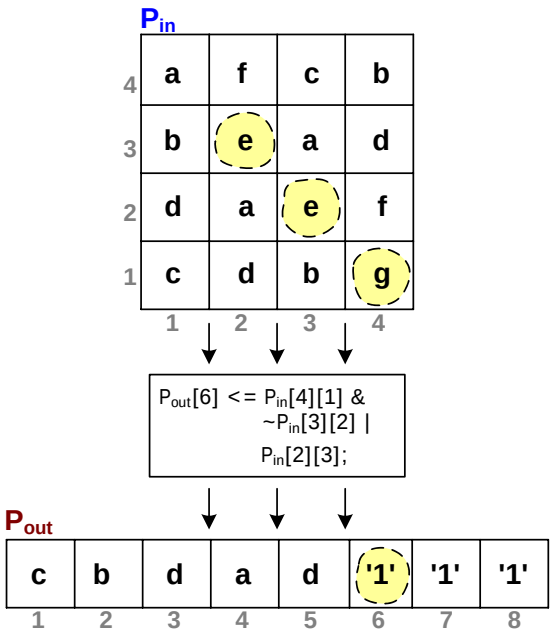

Abbildung 3.3: Zuweisung nach Anwendung von booleschen Funktionen auf eingehende Bits

Die beiden Abbildungen beschreiben jeweils eine Zuweisung der Bits aus dem Datenpaket (P_{in}) der sendenden Task auf das Datenpaket der empfangenden Task (P_{out}). Hinter den Paketnamen kann der "adressierte" Bereich innerhalb des Datenpaketes in eckigen Klammern angegeben werden. In der ersten Klammer werden die Spalten des Datenpakets spezifiziert. Dabei kann es sich um eine einzelne Spalte wie bei $P_{out}[5]$ (5. Spalte) handeln, oder um ein Intervall wie bei $P_{out}[1{:}4]$ (von der 1. bis zu 4. Spalte). Besteht das Datenpaket aus mehreren Zeilen, so wird in der zweiten Klammer – analog zu den Spalten - der Zeilenbereich angegeben. Die Schnittmenge der Spalten- und Zeilenbereiche definiert "adressierte" Zellen (Bits) des Datenpaketes. Die Notationsart erinnert stark an die MatLab[2]- Schreibweise für Matrizenfunktionen und wurde bewusst für die Definition der *Mapping Functions* gewählt, denn auf einer höheren Abstraktionsebene stellen die Datenpakete nichts anderes als zweidimensionale Matrizen dar.
Wie in der Abbildung 3.2 dargestellt, bildet die 1. Abbildungsvorschrift der *Mapping Function* die erste Spalte des P_{in} auf die ersten vier Bits des ausgehenden Datenpaketes ab. Anschließen wird in der 2. Abbildungsvorschrift das Bit aus der zweiten Spalte und ersten Zeile auf das fünfte Bit abgebildet. Die etwas kürzere und elegantere Alternativschreibweise für die 1. und 2. Abbildung lässt sich durch das Konkatenationszeichen ("+") zwischen den beiden P_{in}-Ausdrücken erzielen. Die entsprechende Regel sieht dann

[2]MatLab® ist eine höhere Programmiersprache um technisch-wissenschaftliche Probleme und dazugehörige Lösungen in mathematischer Notation auszudrücken.

folgendermaßen aus: $P_{out}[1:5] \Leftarrow P_{in}[1][1:4] + P_{in}[2][1]$.
Im folgenden Beispiel 3) ist eine Zuwesung konstanter Werte an ein Datenpaket definiert. Die Abbildungsvorschrift weist den letzten beiden Bits des P_{out} einen konstanten Wert zu:

3) $P_{out}[7:8] \Leftarrow \{'1', '1'\}$; $//\equiv (P_{out}[7] \Leftarrow '1';\ P_{out}[8] \Leftarrow '1')$;

Anstatt die Werte einzeln zuzuweisen, können diese – durch ein Komma getrennt – in einem Ausdruck in geschweiften Klammern angegeben werden. Auch die Zuweisung konstanter Werte über mehrere Zeilen eines Datenpaketes ist möglich. Betrachten man den Fall der Umkehrung der Senderichtung, so wäre P_{out} das Datenpaket des parallelen Protokolls und P_{in} das des Seriellen (siehe Abbildung 3.4). Um nun den ersten drei Zeilen eine Konstante zuzuweisen lässt sich das folgendermaßen ausdrücken: $P_{out}[1:4][1:3] \Leftarrow \{\{'1','0','1','1'\}, \{'1','0','0','1'\}, \{'0','0','1','1'\}\}$.

Als Beispiel für die Anwendung der booleschen Funkionen auf die einzelnen Bits des eingehenden Datenpaketes kann eine Regel zum Beispiel so definiert werden:

4) $P_{out}[6][1] \Leftarrow P_{in}[4][1]$ & ˜$P_{in}[3][2]$ | $P_{in}[2][3]$;

Die Regel 4) definiert die Abhängigkeit des 6. Bits in P_{out} von dem Wert des booleschen Ausdrucks mit drei Variablen und drei booleschen bitweise Operatoren (& ≡ and, ˜≡ not, | ≡ or). Mit dem Beispielwert e = '1' ergibt sich in der Abbildung 3.3 somit der Wert '1' für das $P_{out}[6]$.

3.2 Komplexere Abbildungsregeln

Für die Beschreibung komplexerer Abbildungen kann der Systemdesigner endliche Zustandsautomaten zur Datenmanipulation verwenden. Vergleichbar mit höheren Programmiersprachen bieten die Abbildungssprache dafür einen *if-else* Konstrukt an. Die *if-else* Anweisungen lassen sich auch entsprechen beliebig ineinander verschachteln. Die einzelnen Blöcke werden dabei – vergleichbar mit JAVA – durch die geschweiften Klammern "umschlossen". So ist die Definition folgender Regel möglich:

5) bit savedValue = '1';

6)
```
if ((P_in[1:4][3] == {'0','0','1','1'}) && (savedValue != '1')) {
    P_out[1:4] ⇐ P_in[1:2][3] + {'0','0'};
    savedValue = '1'
} else {
    P_out[1:4] ⇐ P_in[1:4][3];
    savedValue = '0';
}
```

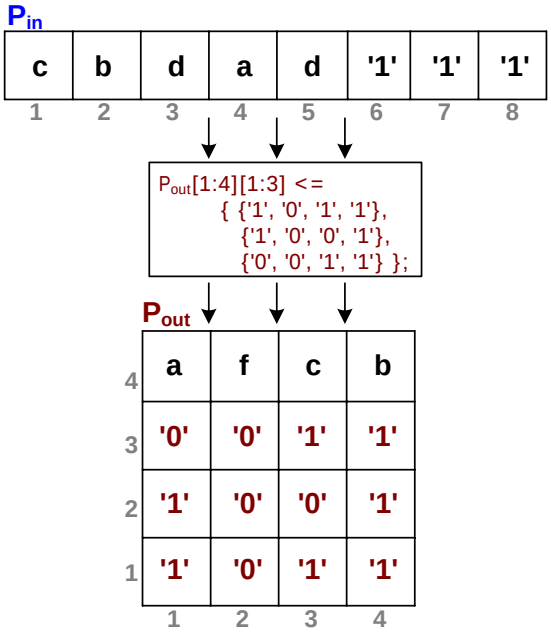

Abbildung 3.4: Zuweisung “mehrdimensionaler” Konstanten

Die Regel 6) beschreibt die Abhängigkeit des ausgehenden Datenpaketes von den Werten des eingehenden Paketes und einer Variable (savedValue). Hat die vorher definierte und mit ’1’ initialisierte Bit-Variable (Regel 5)) einen Wert ungleich ’0’ und sind die Datenbits des Datenpaketes P_{in} gleich dem Bitvektor ’0011’, so werden die ersten vier Bits des P_{out} auf ’be00’ gesetzt. Andernfalls werden ihnen die Bits des P_{in} in der 3. Zeile zugewiesen (siehe Abbildung 3.2).
Während der Synthese werden die booleschen Ausdrücke der *if-else*-Bedingungen als Zustandsübergangsbedingungen und die Zuweisungen als Zustandsausgaben umgesetzt (siehe Kapitel 4). Somit lassen sich durch die bedingten Zuweisungen FSMs in Abhängigkeit von Paketwerten bzw. Variablen beschreiben.

Ein weiteres wichtiges Sprachkonstrukt ist die Angabe von *Instanzen* eines Datenpaketes. Für jedes Datenpaket in einer Abbildungsregel kann zusätzlich zu den Dimensionen auch dessen *Instanz* mit angegeben werden. Dies ermöglicht es, Regeln zu definieren, um mehrere “kleine” Datenpakete auf ein “großes”, oder umgekehrt ein “großes” Datenpaket auf mehrere “kleine” abzubilden. Die **vollständige** Abbildung der Datenbits aus dem P_{in} in der Abbildung 3.3 auf das P_{out} ist beispielsweise nicht möglich, da das P_{out} mit 8 Datenbits halb so klein wie das P_{in} ist. Um dennoch alle Daten des P_{in} übertragen zu können, müssen diese in zwei ausgehende Datenpakete (P_{out}) abgebildet werden. Analog dazu müssen in die andere Richtung zwei kleinere Datenpakete empfangen werden, um ein großes vollständig versenden zu können (siehe Abbildung 3.4). Da es sich jedoch dabei um *den selben* Paketbezeichner handelt, wird die Mehrzahl (die Instanzen) durch einen Index hinter dem Paketnamen ausgedrückt. Um also das P_{in} aus der Abbildung 3.5 kom-

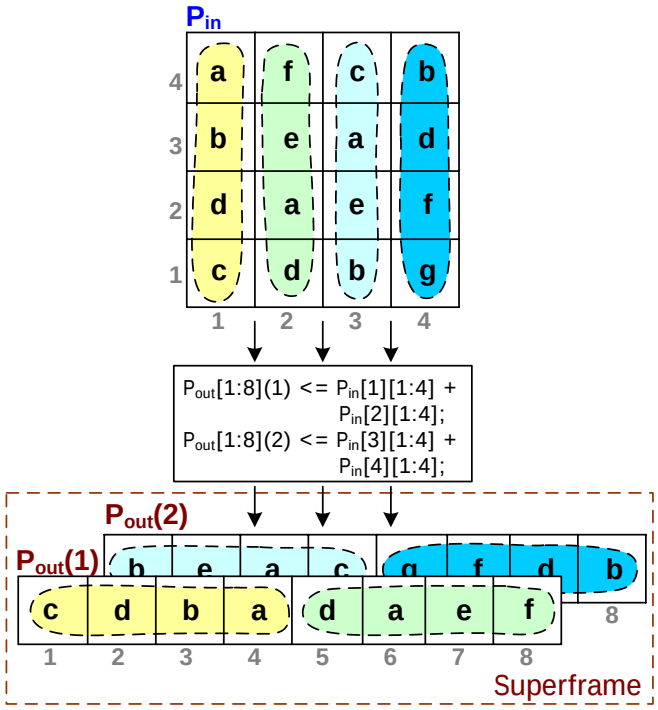

Abbildung 3.5: Abbildungen mit Instanzen

plett zu übertragen, kann die entsprechende *Mapping Function* folgendermaßen definiert werden:

7) $P_{out}[1{:}8](\mathbf{1}) \Leftarrow P_{in}[1][1{:}4] + P_{in}[2][1{:}4];$
$P_{out}[1{:}8](\mathbf{2}) \Leftarrow P_{in}[3][1{:}4] + P_{in}[4][1{:}4];$

Die Datenbits des P_{in} werden spaltenweise auf die zwei Instanzen des P_{out} abgebildet. Somit lassen sich mit dem *IFD-Mapping* auch Datenpakete unterschiedlicher Größen transformieren.
Die Nutzung von Paketinstanzen ist nicht unproblematisch für die IFB interne Kommunikation. Werden mehrere Instanzen eines Datenpaketes benutzt, so impliziert es die Mehrfachübertragung der Frames in denen die Datenpakete sich befinden. Wie bereits in Kapitel 2.3 auf Seite 17 erklärt, stellen die Frames die übertragende Einheiten zwischen dem PH und SH dar. Der Scheduler im IFB lässt aber nur die einmalige Ausführung der *Eingabe*, *Verarbeitung* und *Ausgabe* (EVA) pro Kommunikationszyklus zu. Um dennoch mehrere Instanzen eines Frames übertragen zu können, erfolgt eine Modifikation der Stubs. Hierfür führt die ProtocolFSM den Grundblock in dem sich der Frame befindet so oft hintereinander aus, wie es die Anzahl der Instanzen erfordert. Dieses Vorgehen der "Vervielfachung" eines Grundblocks ist jedoch nur dann erlaubt, wenn ein Grundblock eine Selbsttransition besitzt. Der durch die Wiederholung entstandene Frame wird als *Superframe* bezeichnet und erlaubt es, mehrere Instanzen eines Datenpaketes zu lesen

bzw. zu schreiben, damit das EVA-Schema eingehalten wird [3].
So werden in dem Beispiel aus der Abbildung 3.5 die beiden Instanzen des P_{out} als ein *Superframe* versendet.
Werden für die Definition der *Mapping Function* keine Paketinstanzen benötigt, so kann, wie in den Beispielregeln 1) - 6), auf die Eingabe verzichtet werden. In diesem Fall wird jedes Datenpaket als *eine* Instanz interpretiert.

Um oft benötigten Funktionalitäten nicht immer neu definieren zu müssen, bieten Abbildungssprache eine Importmöglichkeit. Eine bereits definierte und in einer Datei gespeicherte *Mapping Function* kann mit dem Schlüsselwort "import" in eine beliebige Stelle in der neuen *Mapping Function* eingefügt werden. Dieses "Inlining" ermöglicht die Wiederverwendung der bereits definierten Abbildungsregeln.

Mit den bisher vorgestellten Konstrukten der Abbildungssprache ist der Systemdesigner bereits in der Lage alle gewünschten Datenabbildungen zu definieren. Um den vollständigen Umfang der Sprache zu definieren, wird in Kapitel 3.4 die Grammatik in EBNF vorgestellt. Nach der Definition der *Mapping Function* kann anschließen eine syntaktische und semantische Überprüfung durchgeführt werden.

3.3 Syntax- und Semantikanalyse

Für die Kontrolle der syntaktischen Korrektheit der definierten *Mapping Functions* stellt das IFD-Mapping eine Syntax- und Semantikanalyse bereit. Hierfür wurde das Java-Tool "IFS-Editor" um die GUI für die Eingabe und Analyse (frontend) der *Mapping Functions* erweitert (siehe Kapitel 5).
War die syntaktische Analyse erfolgreich, das heißt, entspricht die Grammatik der Quellsprache, so wird anschließend die semantische Analyse durchgeführt. Andernfalls muss der Systemdesigner die Fehler korrigieren.

Die semantische Analyse überprüft die definierten Regeln auf die logische Konsistenz. Folgende Punkte werden dabei untersucht:

- *Zuweisungsoperatoren*
  Es stehen zwei Operatoren für Zuweisungen zur Verfügung. Dabei darf für Zuweisungen an eine Variable nur der "="-Operator benutzt werden. Für die Zuweisungen an einen Datenpaket ist der "⇐"-Operator zuständig.

int savedValue ⇐ 199;	falsch
Pout[1] ⇐ '1';	richtig

- *Datentypen*
  Bei einer Zuweisung an eine Variable bzw. einen Datenpaket oder Variableninstanziierung müssen die Datentypen der linken Seite eines Ausdruckes mit dem Datentyp des Ausdrucks der rechten Seite übereinstimmen. Ein Typecasting wird nicht unterstützt. Dasselbe gilt für die Vergleichsoperationen in Bedingungen.

int savedValue = true;	falsch
P_{out}[1] $\Leftarrow$ savedValue;	falsch
boolean boolValue = false;	richtig

- *Variablen*
  Vor der Erstbenutzung der Variablen, müssen diese definiert werden.
- *Dimensionen*
  Während einer Zuweisung an eine Variable oder ein Datenpaket müssen die Dimensionen des linken Ausdrucks mit dem Ausdruckswert der rechten Seite übereinstimmen. Außerdem dürfen Bereiche außerhalb eines Datenpaketes nicht "adressiert" werden.

$P_{out} \Leftarrow P_{in}$;	richtig, falls gleiche Paketgrößen
P_{out}[1:3][1:2] $\Leftarrow$ P_{in}[1:3];	falsch
bit[][] savedValue = {{'1','0'}, {'1', '1'}};	richtig
int savedValue = {19, 30};	falsch

- *Datenpaketrichtung*
  Es dürfen nur eingehende Datenpakete auf ausgehende Pakete abgebildet werden. Das heißt, auf der linken Seite eines Ausdrucks darf kein eingehendes Datenpaket und auf der rechten Seite kein ausgehendes vorkommen.

P_{in}[1:3][1:2] $\Leftarrow$ P_{out}[1:3][1:2];	falsch

- *Boolesche Operationen*
  Boolesche Operationen wie OR ('|'), AND ('&'), NOT ('!'), XOR ('^') dürfen nur auf Daten vom Typ 'bit' angewand werden.
- *Definitionsreihenfolge*
  Wie bei Variablen so auch bei den Datenpaketen überschreibt die neue Zuweisung die eventuell vorher definierte Abbildungsregel. Das heißt, dass in einer *Mapping Function* die jeweils zuletzt definierte Abbildung auf ein Bit Vorrang vor anderen Abbildungsdefinitionen hat.

Wird durch die definierten *Mapping Functions* eine der oben genannten Kriterien verletzt, so führt das zum Abbruch der Analyse.
Eine weitere wichtige Voraussetzung für die erfolgreiche Analyse der *Mapping Functions* ist die konfliktfreie Datenabbildung. Konfliktfrei bedeutet, dass keine Abbildungen definiert wurden, die zur Verletzung der Datenkausalität während der Ausführung des EVA Zyklusses oder zu einer Verklemmung (Deadlock) der Datenübertragung führen.

Kausalität des Datenverkehrs

Die Control Unit im IFB steuert den Datenverkehr zwischen PH und SH. Neben der Datenbußarbitrierung für die PH-Modes und SH-Modes, sorgt sie durch ein Scoreboard für

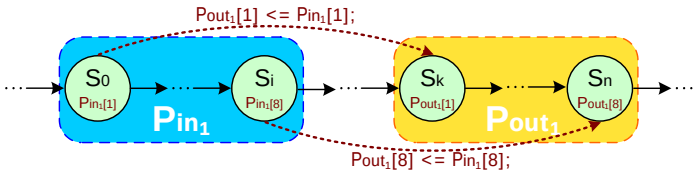

Abbildung 3.6: Datenpakete der ProtocolFSM und bitweise Abbildung der Daten

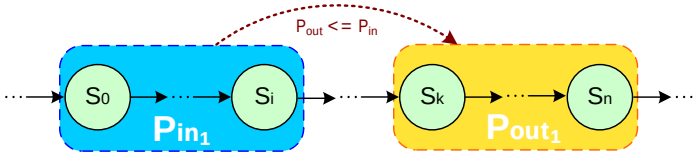

Abbildung 3.7: Abbildungsregel auf Datepaketenebene zur Abbildung 3.6

die Einhaltung der Kausalität während des Datenverkehrs. Daten dürfen erst gesendet werden, wenn sie vorher vollständig empfangen wurden.
In der Abbildung 3.6 ist der Ausschnitt einer ProtocolFSM dargestellt. Die Zustände s_0 bis s_i repräsentieren von einer Task gesendete Daten als ein Datenpaket P_{in}. Die Zustandssequenz s_k bis s_n stellt von der Task zu empfangenden Datenpaket P_{out} dar. Beide Datenpakete sind farblich hervorgehoben. Eine mögliche Abbildungsvorschrift der Daten zwischen den zwei Datenpaketen kann beispielsweise durch die bitweise Zuweisung der eingehenden auf ausgehende Daten definiert werden. Die gestrichelten Pfeile repräsentieren dabei die Abbildungsrichtung, und die Pfeilbeschriftungen die jeweilige Abbildungsvorschrift. Unter der Voraussetzung, dass das P_{in} und P_{out} gleich groß sind, lässt sich die Abbildungsregel auch direkt durch die Datenpaketzuweisung ($P_{out} \Leftarrow P_{in};$) beschreiben (vergleiche Abbildung 3.7).
Das Beispiel stellt eine semantisch gültige *Mapping Function* dar, und erfüllt das EVA Schema indem die Daten erst nach deren Empfang gesendet werden.

Werden nur die Datenpakete und deren Abhängigkeiten untereinander betrachtet, so lassen sich diese in Form eines Datenpaketgraphen visualisieren. Die Datenpakete stellen dabei die Knoten des Graphen dar. Die Beziehungen zwischen den Knoten werden dabei

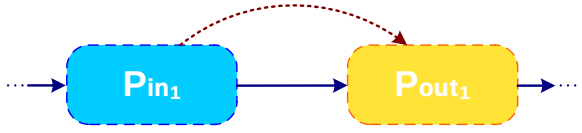

Abbildung 3.8: Datenpaketgraph zur Abbildung 3.6

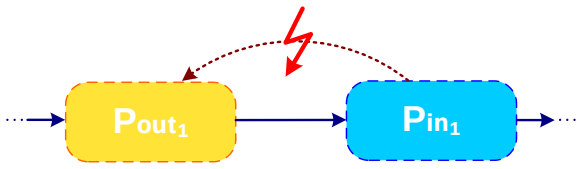

Abbildung 3.9: Verletzung der Kausalität

durch die Abhängigkeit der Datenpakete einerseits und durch die Zustandsübergänge der ProtocolFSM andererseits, dargestellt (siehe Abbildung 3.8). Der *Datenpaketgraph* kann für die Prüfung der *Mapping Functions* auf Einhaltung des EVA Prinzips genutzt werden. Hierfür muss der Graph auf die Existenz von Zyklen untersucht werden. Existiert mindestens ein solcher Zyklus, so verletzt die *Mapping Function* die Kausalität der Datentransformation.

Ein Szenario für die Verletzung der Kausalität ist in der Abbildung 3.9 dargestellt. Durch das Vertauschen der Datenpaktreihenfolge in der ProtocolFSM aus der Abbildung 3.6 führt nun die Abbildungsregel $P_{out} \Leftarrow P_{in}$ zu einem Verstoß gegen das EVA Prinzip. Wegen der Rückwärtskante von P_{in} nach P_{out} ist im Datenpaketgraphen ein Zyklus entstanden. Die Ausgabe P_{out} der noch nicht empfangenen Daten P_{in} ist nicht möglich.

Deadlocks

Es existieren Szenarien, in denen die Abbildungen der *Mapping Function* zur Verklemmungen (*Deadlocks*) der IFB internen Datenübertragung führen können. Werden z.B. Abbildungsregeln zwischen mehr als einer ProtocolFSM definiert, und beschreiben diese die gegenseitige Abhängigkeit mehrerer Pakete, so darf im resultierenden *Datenpaketgraph* ebenfalls kein Zyklus entstehen. Andernfalls führt die definierte *Mapping Function* zu einer gegenseitigen Verklemmung der Protokollausführung.

Die Abbildung 3.10 zeigt die Abhängigkeiten zwischen den Datenpaketen zweier ProtocolFSMs in Form eines Datenpaketgraphen. Die ausgehenden Datenpakete beider Protokolle sind jeweils von den eingehenden Datenpaketen des anderen Protokolls abhängig. Die *Mapping Function* kann in diesem Fall beispielsweise aus folgenden zwei Abbildungsregeln definiert worden sein:

$P_{out_1} \Leftarrow P_{in_2}$;
$P_{out_2} \Leftarrow P_{in_1}$;

Obwohl die so definierten Abbildungen zu einer gegenseitige Datenabhängigkeit führen, sind sie bezüglich der Verklemmung unkritisch. Desgleichen gilt für das in der Abbil-

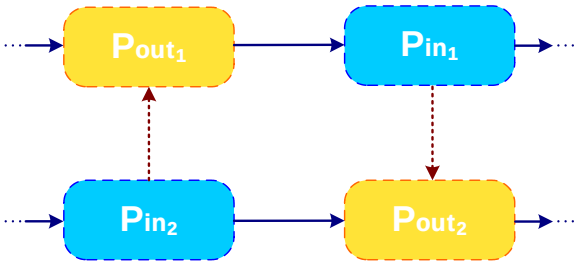

Abbildung 3.10: Verklemmungsfreie Datenabbildung (2 Grundblöcke)

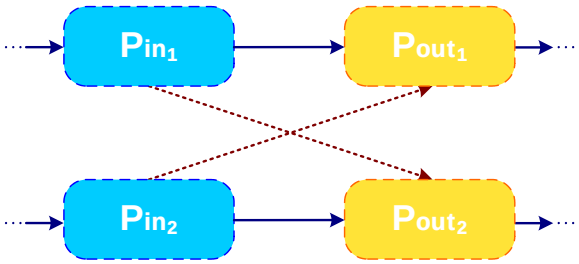

Abbildung 3.11: Verklemmungsfreie Datenabbildung (2 Grundblöcke)

dung 3.11 dargestellte Szenario. Beide Datenpaketgraphen sind kreisfrei, folglich auch Verklemmungsfrei.
Im Gegensatz dazu, ist der Datenpaketgraph in der Abbildung 3.12 nicht kreisfrei. So bildet der Pfad $P_{in_1} \rightarrow P_{out_2} \rightarrow P_{in_2} \rightarrow P_{out_1} \rightarrow [P_{in_1}]$ einen Zyklus. Die zwei oben definierten Abbildungsregeln würden hier zu einem *Deadlock* führen. Denn während das erste Protokoll (oben) auf das Datenpaket P_{in_2} wartet ($P_{out_1} \Leftarrow P_{in_2}$), muss das zweite (unten) seinerseits auf das Datenpaket P_{in_1} aus dem ersten Protokoll warten ($P_{out_2} \Leftarrow P_{in_1}$).

Um die Graphen auf die Kreisfreiheit zu untersuchen, kann der *Depth-first-search* (DFS) Algorithmus verwenden werden (siehe [11]). DFS ist ein effizientes Verfahren, um Informationen über eine Graphen zu gewinnen. Die Laufzeit des Algorithmus ist $\Theta(V + E)$, wobei V für die Anzahl der Knoten und E für die Anzahl der Kanten in einem Graphen stehen.

3.4 Grammatik der Abbildungssprache

Für die formale Beschreibung der Abbildungssprache wurde im Rahmen dieser Studienarbeit die Grammatik in der Erweiterten Backus-Naur-Form verwendet. Die Grundlagen dafür wurden in Kapitel 2.4 vorgestellt. In diesem Kapitel wird nun die Grammatik anhand von Beispielen aus dem vorangegangenem Kapitel vorgestellt.

Jede *Mapping Function* eines IFD-Mappings beinhaltet mehrere Abbildungsvorschriften. Des Weiteren können am Anfang jeder *Mapping Function* optional externe "Funktionsmodule" eingebunden werden. Diese Module stellen vorher definierte und für deren Wiederverwendung entwickelte *Mapping Functions* dar. Somit kann die *Mapping Function* als eine Menge aus externen und internen Abbildungsvorschriften beschrieben werden. In EBNF lässt sich dafür folgende Produktionsregel mit der Startvariable *MappingFunction* definieren:

MappingFunction = {ImportDeclaration} {MappingDeclaration};

Ein Import wird durch das Schlüsselwort "import" gefolgt von einem Bezeichner definiert.

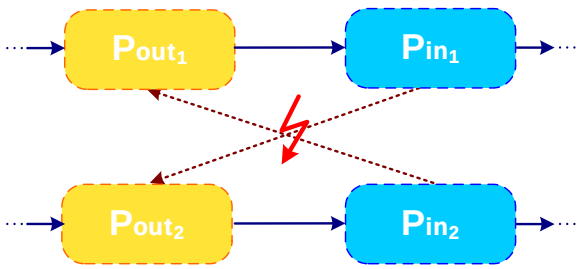

Abbildung 3.12: Beispiel eines Deadlocks (2 Grundblöcke)

Der Bezeichner ist in diesem Fall eine externe *Mapping Function*. Die Produktionsregel sieht daher folgendermaßen aus:

ImportDeclaration = 'import' Identifier;

Identifier (Bezeichner) kann eine beliebige Zeichenfolge aus Buchstaben und Zahlen sein, jedoch muss das erste Zeichen ein Buchstabe sein. Die Buchstaben und Zahlen lassen sich in einer Produktionsregel einfach durch die Angabe aller möglichen Alternativen definieren:

Identifier = Letter {Letter | Digit};
Letter = 'A' | 'B' | 'C' | ··· | 'Z' | 'a' | 'b' | 'c' | ··· | 'z';
Digit = '0' | '1' | '3' | ··· | '9';

Die rechte Seite der letzten beiden Produktionsregeln bestehen nur aus Terminalen und kann daher nicht durch andere Symbole ersetzt werden. Zur Vereinfachung werden nun zuerst die Produktionsregeln gleichen Typs vorgestellt.
Für die Definition einer Variablen muss der Datentyp mit angegeben werden. Das IFD-Mapping unterstützt drei primitive Datentypen: "boolean", "integer" und "bit". Zusätzlich dazu können Arrays aus maximal zwei Dimensionen definiert werden. Ein Zweidimensionales Array vom Typ "bit" kann dann folgendermaßen definiert werden: *bit[][]*. Die entsprechende Produktionsregel dazu:

VariableType = ('boolean' | 'int' | 'bit') ['[]'] ['[]'];

Für die Wertzuweisungen an eine Variable oder ein Datenpaket werden unterschiedliche Zuweisungsoperatoren benutzt. Für Variablen muss ein einfaches Gleichheitszeichen ("="), und für die Datenpakete eine Kombination aus einem Linkspfeil und Gleichheitszeichen ("<="). Daher wird der Zuweisungsoperator in EBNF wie folgt angegeben:

AssignmentOperator = '<=' | '=';

Analog dazu werden die Wertebereiche für die einzelne Datentypen definiert. Für die Zuweisung an eine "bit"-Variable stehen zwei Werte zur Verfügung: "1" und "0". Für "boolean" sind es "true" und "false". Ein "integer"-Literal kann durch eine Folge aus Zahlen (*Digit*) ohne der führenden "0" definiert werden:

Literal = IntegerLiteral | BooleanLiteral | BitLiteral;
IntegerLiteral = (Digit - '0') {Digit};
BooleanLiteral = 'true' | 'false';
BitLiteral = "'('1' | '0')"';

Jedes Datenpaket innerhalb des IFS-Flows besitzt einen eindeutigen Bezeichner. Dieser wird während der Generierung des Datenpaketes dynamisch vergeben, und hilft dem Designer des Systems die Datenpakete unterscheiden zu können. Die Bezeichnung eines Paketes besteht aus einem Präfix gefolgt von einer Id. So haben alle ausgehenden Datenpakete den Präfix "OP_" (*outgoing package*), und alle eingehende "IP_" (*incoming package*). Durch die Kombination von einem Präfix und der Id, kann der Systemdesigner während der Definition der *Mapping Functions* nicht nur die Datenpakete eindeutig unterscheiden, sondern anhand der Bezeichnung auch auf deren Richtung schließen. Die Syntax für die Datenpaketbezeichnung wurde in EBNF daher durch folgende Produktionsregel festgelegt:

ProtocolPackage = ('IP_' | 'OP_') IntegerLiteral;

Um in der Abbildungsregel einen bestimmten Bereich innerhalb von einem Datenpaket "adressieren" zu können, muss in eckigen Klammern die Spalte und die Zeile angegeben werden (vergleiche Datenabbildung 1 auf Seite 27). Die Zeilenangabe ist jedoch optional. Bei Spalten und Zeilen kann es sich dabei auch um Intervalle handeln. Durch einen Doppelpunkt getrennt, gibt die erste Zahl den Anfang und die letzte Zahl das Ende des Intervalls an. Somit ergeben sich folgende zwei Produktionsregeln:

ProtocolPackageSuffix = '[' IntInterval ']' ['[' IntInterval ']'];
IntInterval = IntegerLiteral [':' IntegerLiteral];

Für einfache und bedingte Zuweisungen an die Variablen oder Datenpakete können komplexe Ausdrücke definiert werden. So kann die *if*-Bedingung bzw. die rechte Seite einer Zuweisung (nach dem Zuweisungsoperator) aus mehreren – durch logische Operatoren verknüpften - booleschen Ausdrücken bestehen. Die Sprache unterstützt neben der Negation ("!") zwei weitere logische Operatoren, den *oder*- und *und*-Operator.

ConditionalOrExpression = ConditionalAndExpression
{'||' ConditionalAndExpression};
ConditionalAndExpression = InclusiveOrExpression {'&&' InclusiveOrExpression};

Die beiden Produktionsregeln ermöglichen es beliebige boolesche Ausdrücke der Form (A && B || C || D && $E \cdots$) zu definieren. Ein Beispiel für die Verwendung eines *und*-Operators in einer bedingten Zuweisung wurde bereits mit der Datenabbildung 6) auf Seite 29 vorgestellt.
Des Weiteren erlaubt die Sprache auch die (bitweise) Anwendung von boolesche Operatoren auf Bitfelder. Neben den in Kapitel 3.1 vorgestellten *and*, *not* und *or* Operatoren, kann zusätzlich dazu der *xor* Operator verwendet werden.
Für Vergleichsoperationen in booleschen Ausdrücken werden zwei Operatoren bereitgestellt: *'=='* (gleich) und *'!='* (ungleich).
Im Vergleich zu anderen Programmiersprachen hat der Plusoperator im IFD-Mapping

eine andere Semantik. Als Additionsoperator kann das Pluszeichen nur für ganze Zahlen (*int*) benutzt werden. Ansonsten dient er zur Konkatenation zweier Bits bzw. Bitfelder. Ein Beispiel der Konkatenation von zwei Teilpaketen und anschliessenden Zuweisung des Ergebnisses an ein ausgehendes Datenpaket wurde in Kapitel 3.1 bereits vorgestellt ($P_{out}[1:5] \Leftarrow P_{in}[1][1:4] + P_{in}[2][1]$).

```
InclusiveOrExpression      = ExclusiveOrExpression {'|' ExclusiveOrExpression};
ExclusiveOrExpression      = AndExpression {'^' AndExpression};
AndExpression              = EqualityExpression {'&' EqualityExpression};
EqualityExpression         = AdditiveExpression {('==' '!=') AdditiveExpression};
AdditiveExpression         = UnaryExpression {'+' UnaryExpression};
UnaryExpression            = '~' UnaryExpression | '!' UnaryExpression |
                           | PrimaryExpression;
```

Die vorgestellten Operatoren können auf Datenpakete, Variablen, Literale, Arrays aus Konstanten (auch zweidimensionale) und weitere komplexe Ausdrücke angewandt werden. Ein Array wird durch eine Folge von durch Kommas getrennten Konstanten (*Literal*) definiert. Zusätzlich wird die Folge durch geschweifte Klammern umschlossen. Das zweidimensionale Array wird als eine Arrayfolge – wieder durch Kommas getrennt – in geschweiften Klammern repräsentiert. Die Beispielnotation eines Arrays mit 3 Spalten und 2 Zeilen könnte dann folgendermassen aussehen: {{'1','0','1'}, {'0','0','1'}}.

```
PrimaryExpression          = PackageOrVariable | Literal | ConstantArray |
                           | '{' ConstantArray {',' ConstantArray} '}' |
                           | '(' ConditionalOrExpression ')';
ConstantArray              = '{' Literal {',' Literal} '}';
```

Aus der Nichtterminale *PackageOrVariable* können, wie der Name schon sagt, entweder ein Datenpaket oder einer Variable abgeleitet werden. Die Produktionsregel für eine Variable wurde weiter oben bereits definiert (*Identifier*), und die Syntax für den Datenpaketbezeichner wurde mit dem Nichtterminalsymbol *ProtocolPackage* eingeführt. Zusätzlich zum Paketnamen kann in einer Abbildungsregel die Instanz (optional) des Datenpaketes in runden Klammern, sowie die Dimensionen in eckigen Klammern, mit angegeben werden.

```
PackageOrVariable          = ProtocolPackage ['(' IntegerLiteral ')']
                           [ProtocolPackageSuffix] | Identifier;
```

Die Abbildungsvorschriften sind entweder einfache Variablendefinitionen, Zuweisungen an die Variablen bzw. Datenpakete, oder bedingte Anweisungen. Jede Variablendefinition und Zuweisung muss dabei mit einem Semikolon enden. Die Abbildungsvorschriften können zu besseren Übersicht in geschweiften Klammern beliebig gruppiert werden. Daher wird die Produktionsregel für die Nichtterminale *MappingDeclaration* dementsprechend so definiert:

```
MappingDeclaration  = VariableDeclarator ';' | Statement;
Statement           = Block | IfStatement | Assignment ';' ;
Block               = '{' {MappingDeclaration}' }';
```

Durch das Nichtterminalsymbol *VariableDeclarator* wird die Syntax der Variablendefinitionen einer *Mapping Function* festgelegt. Eine Variable wird durch den Variablentyp gefolgt von einem Bezeichner definiert. Optional kann die Variable mit einem Anfangswert initialisiert werden. Der Anfangswert kann von einem einfachen Typ bis zum komplexen Ausdruckswert reichen. So wurde Beispielsweise die *bit*-Variable *"savedValue"* in der Beispielabbildung 6 auf Seite 29 definiert und mit einem Anfangswert initialisiert.

```
VariableDeclarator  = VariableType Identifier ['=' ConditionalOrExpression];
```

Aus der Nichtterminale *Statement* werden weitere Produktionsregeln für die Zuweisungen und dir *if*-Anweisung abgeleitet. Eine Zuweisung beginnt mit dem Bezeichner einer Variable oder Datenpaketes, gefolgt von dem entsprechenden Zuweisungsoperator. Anschließend wird, ähnlich der Initialisierung einer Variablen, ein Ausdruck angegeben.

```
Assignment          = PackageOrVariable AssignmentOperator
                      ConditionalOrExpression;
```

Die *if*-Anwesung beginnt mit dem Schlüsselwort "if" gefolgt von einem geklammerten Ausdruck. Nach der schliessenden Klammer muss eine Anweisung (*Statement*) folgen. Der "sonst"-Fall einer *if*-Anweisung wird mit dem Schlüsselwort "else" eingeleitet, der wieder mit einer Anweisung enden muss.

```
IfStatement         = 'if' '('ConditionalOrExpression')' Statement
                      ['else' Statement];
```

Eine Übersicht über die oben vorgestellten Produktionsregeln befindet sich im Anhang dieser Arbeit.

4 IFD-Mapping Anwendung

Das IFD-Mapping stellt neben den Schnittstellenbeschreibungen (IFDs) und der Zielplattformbeschreibung (TPD) eine weitere wichtige Eingabe für die Erzeugung des Schnittstellenadapters dar. So wird durch das IFD-Mapping nicht "nur" festgelegt wie die Daten zwischen zwei Protokollen transformiert werden, sondern es liefert für die anwendungsbezogene Schnittstellensynthese die notwendigen Parameter für die Optimierung und Parametrisierung der IFB-internen Komponenten hinsichtlich eines Anwendungsszenarios.

In der Abbildung 4.1 ist der Ablauf des ersten Syntheseschritts im IFS-Flow dargestellt. Das Ziel in dieser Synthesephase ist die Generierung eines zielsprachenunabhängigen

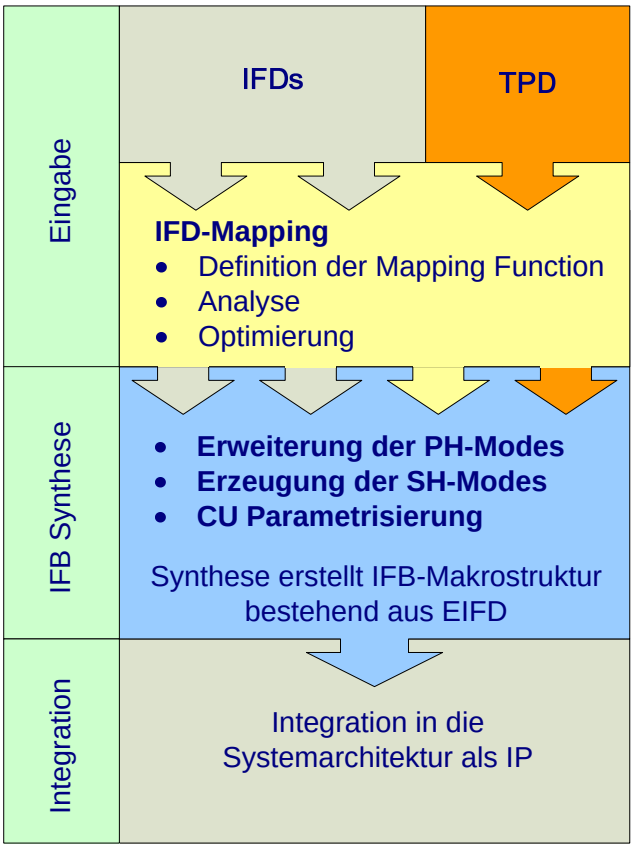

Abbildung 4.1: Erster Syntheseschritt im *Interface Synthesis Design Flow*

Schnittstellenadaptermodells in einem Zwischenformat. Nach der erfolgreichen Synthese des Adapters kann dieser dann als eine IP in die bestehende Systemarchitektur integriert, oder in einem XML-Format exportiert werden.
Die Modellierungsphase ist in drei Schritte aufgeteilt. Die TPD als Ausführungsplattform und vom Designer ausgewählten IFDs der Systemkomponenten stellen zusammen mit dem IFD-Mapping den Input der Synthese dar. Die Protokollsynthese generiert an dieser Stelle anhand der Protokollbeschreibungen aus den IFDs die Stubs im PH. Aus den ProtocolFSMs der Stubs werden die Datenpakete erzeugt, die der Designer mit Hilfe der Mapping Functions aufeinander abbildet. Während der Syntax- und Semantikanalyse werden die in den Abbildungsvorschriften vorkommenden Datenpaketbits für die darauf folgenden Optimierungen markiert. Im Verlauf der Optimierung werden die Datenpakete auf die markierten Datenbits minimiert und die Pakete sowie die Frames entsprechend angepasst.
Im zweiten Schritt der Synthesephase, der eigentlichen IFB Synthese, fliessen die Ergebnisse aus dem IFD-Mapping in den Generierungs- und Optimierungsprozess der IFB-Makrostruktur. Die generierten Hauptkomponenten der IFB-Makrostruktur (die CU, der SH und der PH) liegen anschließend als EIFD (*extended* IFD) vor.
Nach einer erfolgreichen Synthese kann der IFB in die Systemarchitektur als IP integriert werden bzw. als Zielcode erzeugt und in das eigentliche Design eingefügt werden.

4.1 Optimierung

Nach der erfolgreichen Syntax- und Semantikanalyse der *Mapping Functions* werden die Informationen dazu genutzt, die im IFB zu verarbeitenden Datenpakete zu optimieren. Hierzu werden die nicht benötigten Datenbits aus den Datenpaketen entfernt. Die neuen Datenpakete enthalten daraufhin nur noch die Daten, die in den Abbildungsvorschriften “adressiert” worden sind. Dieses Schrumpfen der Datenpakete führt zu einer Verkleinerung des erforderlichen Zwischenspeichers im Sequence Handler. Dadurch kann für die Transformation der Daten genau die Menge an Speicher bereitgestellt werden, die für das festgelegte Anwendungsszenario benötigt wird.
Des Weiteren werden die IFB interne Datenbusse an die erforderliche Grösse angepasst. Es werden genau so viele Leitungen für die Datenübertragung synthetisiert, wie für die Übertragung der Datenpakete notwendig ist.
Weiterhin tragen die optimierten Datenpakete zur kürzeren Übertragungszeiten der Daten bei. Da die Pakete kleiner werden, wird auch weniger Zeit benötigt, um sie zu übertragen. Folglich verkürzen sich die Zeiten, in denen die Protocol Handler den Datenbuss belegen.
Der Verkleinerung der Datenpakete folgt die Neubestimmung der Frames. Die Frames werden anhand der neuen Datenpaketstruktur neu ermittelt (siehe Unterkapitel **Frames** auf Seite 17), und anschließend in die Erweiterung der PH-Modes und Parametrisierung der Control Unit eingesetzt.

4.2 Erweiterung der PH-Modes

Für die Datenübertragung zwischen PH und SH werden die ProtocolFSM der Stubs (PH-Modes) um die dafür notwendige Zustände erweitert. Das IFD-Mapping liefert die Informationen wann und welche Frames übertragen werden.

Abbildung 4.2 zeigt einen beispielhaften PH-Mode, dessen ProtocolFSM die Daten von einer Task empfängt und diese dann in den Speicher des Sequence Handlers (DataReader) schreibt. Durch die Frames im IFD-Mapping kann die ProtocolFSM nun um die für die Frameübertragung und für die Kommunikation mit dem SH-Mode notwendigen Zustände (rot umrandet) erweitert werden. In *Establish Bus_{In} communication* fordert der PH-Mode den Buszugriff an und teilt gleichzeitig mit, um welchen Frame es sich bei der Übertragung handelt (FrameID). Nach der vollständigen Übertragung des Frames wird in *Release Bus_{In} communication* der Datenbus wieder frei gegeben.

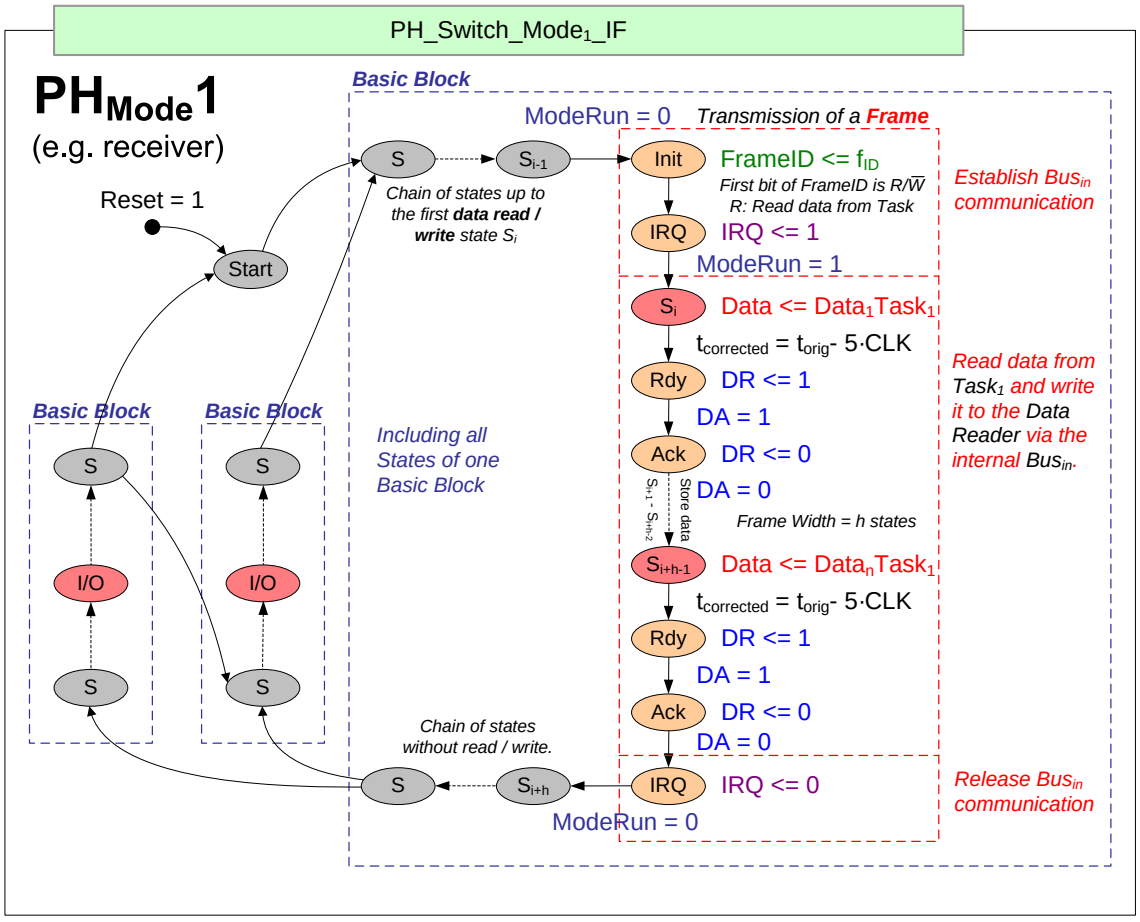

Abbildung 4.2: Modifikation des PH-Modes

4.3 Auswirkungen auf SH

Das IFD-Mapping liefert die für die Generierung der SH-Modes und die Synthese des Zwischenspeichers im SH notwendigen Parameter.

Erzeugung der SH-Modes

Die Abbildungsvorschriften eines IFD-Mappings werden während der *Sequence Handler Synthese* durch die SH-Modes realisiert. Alle Zuweisungen (*mapping equations*) auf ein ausgehendes Datenpaket werden zu einer *Mapping Function* zusammengefasst, und anschließend in einem Zustandsautomat umgesetzt. Dieser Zustandsautomat (SH-Mode) implementiert die im IFD-Mapping definierte Verarbeitung der Protokolldaten.

Ein Template für den SH_{Mode} ist in der Abbildung 4.3 dargestellt. Die *Modify*-Zustände des Automaten werden anhand der *Mapping Function* für das ausgehende Datenpaket erzeugt. Alle *mapping equations* werden dabei als Zustandsausgaben modelliert. Beinhaltet die *Mapping Function* beispielsweise nur "einfache" Zuweisungen, das heisst, ohne eine FSM zur Datenmanipulation zu definieren, so wird nur ein *Modify*-Zustand mit entsprechenden Ausgaben synthetisiert. Als Vorbereitung für die zielplattformspezifische

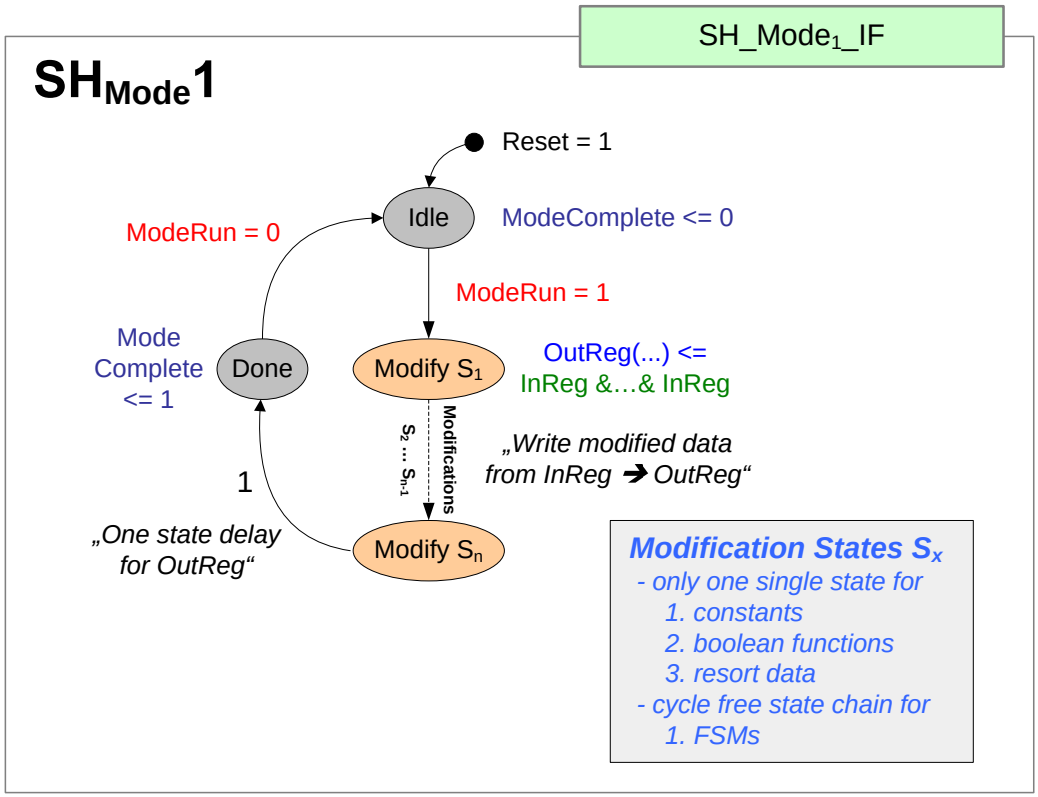

Abbildung 4.3: SH-Mode

IFB-Implementierung im zweiten Syntheseschritt werden Teile des Ableitungsbaums aus der Analysephase als Teil der jeweiligen Zustandsausgabe gespeichert. Diese "Referenzen" auf die Teilbäume dienen als Grundlage für die spätere Codegenerierung in einer beliebigen Zielsprache.
Wurden in der *Mapping Function* FSMs (*if*-Anweisungen) zur Datenmanipulation definiert, so wird mehr als ein Zustand erzeugt. Die *if*- und *else*-Ausdrücke werden jeweils zu Transitionsbedingungen an den Zustandsübergängen umgesetzt. Die Anweisungen aus dem *if*- und *else* Block werden wieder als Moore-Ausgabe in dem jeweiligen Zustand definiert. So zeigt das Beispiel in der Abbildung 4.4 die erzeugte Zustandsfolge eines SH_{Modes} für eine verschachtelte *if-else*-Anweisung. Ähnlich wie oben, werden bei der Erzeugung dieser FSM die zugehörige Ableitungsbaumausschnitte mit gespeichert. In diesem Fall jedoch nicht nur für die Automatenausgaben, sondern auch für die Transitionsbedingungen.

Dedizierter Zwischenspeicher

Für die Synthese des Zwischenspeichers im SH werden die Informationen aus dem IFD-Mapping herangezogen. Da auf den Speicher des Sequence Handlers mehrere SH-Modes parallel zugreifen können, muss die optimale Speichergrösse für die Synthese bestimmt werden (dedizierter Speicher). Das IFD-Mapping liefert mit den ermittelten Frames und Datenpaketen den dafür notwendigen Parameter. Jeder Frame besitzt ausserdem eine eindeutige Id (*FrameId*), die während des Datenaustausches eine exakte Zuordnung zwischen dem Frame und dem Speicher ermöglicht.

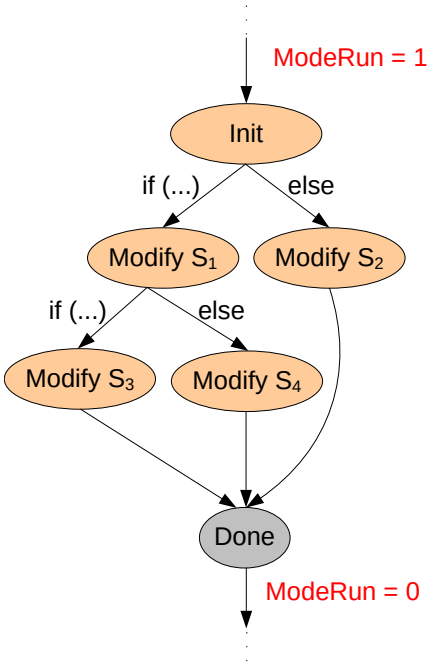

Abbildung 4.4: FSM-Zustände des SH-Mode

4.4 Parametrisierung der CU

Wie bereits im Grundlagenkapitel IFB-Makrostruktur auf der Seite 8 erklärt, ist die *Control Unit* unter anderem für die Arbitrierung der Datenbusse im IFB zuständig. Die Zustände in den beiden Arbitern werden anhand des IFD-Mappings an die Anzahl der auf den Speicherbus zugreifenden Kommunikationskomponenten angepasst. Des Weiteren wird auch das Scoreboard in der CU durch das IFD-Mapping parametrisiert. Für die Aufrecherhaltung der Datenkausalität durch das Scoreboard werden Informationen über die Speichergrösse, die Anzahl der auf den Zwischenspeicher zugreifenden Modes und die Größe bzw. Anzahl die Datenpakete benötigt.

Scoreboard

Zum einen ist das *Scoreboard* in der Control Unit für die Steuerung des Sequence Handlers zuständig. Zum anderen liefert das Scoreboard dem Arbiter die Informationen über den aktuellen Stand der übertragenden Daten. So können die Arbiter mit Hilfe des Scoreboards abfragen, welche Frames geschrieben oder gelesen werden dürfen. Hierfür speichert das Scoreboard in verschiedenen Registern den aktuellen Status der SH-Modes und der Datenpakete. Die Anzahl der Register und die jeweilige Grösse werden durch das IFD-Mapping bestimmt.
In dem "*SH-Mode Complete*" Register wird der aktuelle Status aller SH-Modes gespeichert. Hat ein SH-Mode die Modifikation der Daten erfolgreich abgeschlossen, so wird im Register das dazugehörige Bit auf '1' gesetzt. Zurückgesetzt wird das Bit nachdem das modifizierte Datenpaket (P_{out}) an den Empfänger weitergeschickt wurde. Die Anzahl der SH-Modes (*mapping equations*) im IFD-Mapping legt daher die Dimension des "*SH-Mode Complete*" Registers fest. Zusätzlich dient diese Information, um die Anzahl und die Grösse der "*P_in Complete*" Register festzulegen. Für jeden SH-Mode existiert ein solches Register, das den Status der eingehenden Datenpakete (P_{in}) repräsentiert. Erst wenn alle für den SH-Mode relevanten P_{in}s empfangen worden sind, darf er mit der Verarbeitung beginnen. Demzufolge hängt die Grösse der "*P_in Complete*" Register von der Anzahl der eingehenden Datenpakete ab und wird aus dem IFD-Mapping ermittelt.

Arbiter

Die Datenbuszugriffe im IFB werden von den Schedulern innerhalb der Control Unit gesteuert. Für die beiden unidirektionalen und von einander unabhängigen Datenbusse zwischen dem PH und SH existiert daher jeweils ein Scheduler. Der $Ctrl_{In}$ ist für die Arbitrierung des Datenbusses vom PH zum SH, und der $Ctrl_{Out}$ vom SH zum PH zuständig. In der Abbildung 4.5 ist der Schedule $Ctrl_{In}$ als Moore Automat dargestellt. Der "*Schedule Frame*" Zustand ist der *Idle* Zustand des Automaten. Die Anzahl der "*Grant Bus*" Zustände hängt von der Anzahl an eingehenden Frames ab, und wird an dieser Stelle vom IFD-Mapping bereitgestellt. Ein PH-Mode bewirbt sich um den Datenbus zum SH indem er den entsprechenden *IRQ* setzt und gleichzeitig die *FrameID* mitteilt (vgl. Abbildung 4.2). Falls das Scoreboard meldet, dass der Frame gelesen werden darf, findet

der Zustandswechsel von "*Schedule Frame*" in den entsprechenden "*Grant Bus*" Zustand statt. In dem "*Grant Bus*" Zustand wird der Datenbus für den PH-Mode freigegeben um den Frame zum SH zu übertragen. Nach der erfolgreichen Frameübertragung seitens des PH-Mode ($IRQ <= 0$) wechselt der Scheduler in den "*Assign Frame*" Zustand, wo er die *FrameId* an das Scoreboard senden und den Datenbus wieder frei gibt. Andernfalls wird nur der Datenbus freigegeben (*Release Bus*) und die Übertragung gilt damit als erfolglos.

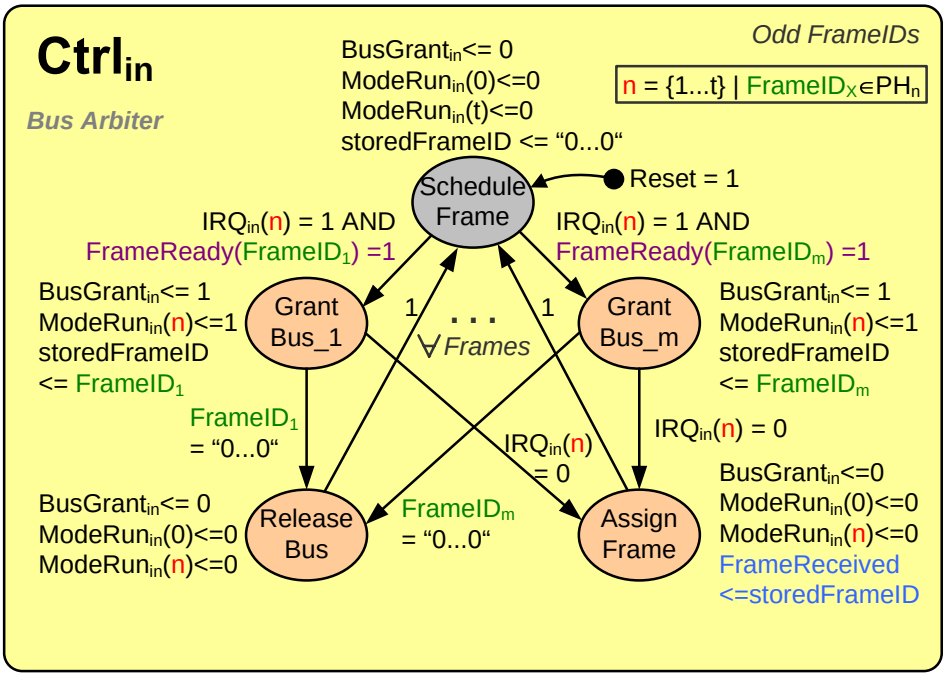

Abbildung 4.5: Scheduler

5 IFD-Mapping Umsetzung

In diesem Kapitel wird die Umsetzung des vorgestellten IFD-Mapping Konzeptes in Java erläutert. Hierfür werden lediglich die wichtigsten Java Datenstrukturen und implementierten Funktionen für die Integration des IFD-Mapping in den bestehenden Syntheseablauf vorgestellt. Zunächst wird der *Mapping Editor* als grafische Oberfläche (GUI) für die Definitionen der Mapping Functions eingeführt. Danach folgen die wichtigsten Klassen mit ihren Methoden, die den Kern des IFD-Mappings bilden. Anschließend wird kurz auf die Generierung des Syntax- und Semantikanalysierers aus der in Kapitel 3.4 vorgestellten Grammatik mit Hilfe des JavaCC Tools eingegangen.

5.1 Integration im IFS-Editor

Der IFS-Editor implementiert die Konzepte des IFS-Flows. Das in Java entwickeltes Tool unterstützt den Designer von der Modellierung der Systemarchitektur über die Synthese der IFBs bis zur Kodegenerierung.
Als Bestandteil des ersten Syntheseschritts im IFS-Flow, der Erstellung der abstakten IFB Instanz, wurde die Eingabe des IFD-Mappings in den *Synthesis Wizard* integriert. Der *Synthesis Wizard* führt den Systemdesigner schrittweise durch alle Phasen der IFB-Synthese. Im ersten Schritt können die Schnittstellenbeschreibungen der zu verbinden Tasks selektiert werden, gefolgt von der Möglichkeit die Zielplattformbeschreibung auszuwählen. Im nächsten Schritt muss der Designer das IFD-Mapping angeben.
Um das IFD-Mapping zu spezifizieren, das heisst, die Abbildungsregeln beschreiben zu können, wurde im Rahmen dieser Studienarbeit ein *Mapping Editor* entwickelt. Der *Mapping Editor* ist eine in Java implementierte GUI, die es dem Systemdesigner ermöglicht Mapping Functions auf einfache Weise zu definieren, zu analysieren und in den Syntheseprozess zu integrieren.

In der Abbildung 5.1 ist der *Mapping Editor* dargestellt. Über die Auswahlboxen in der oberen Leiste können die Schnittstellenbeschreibungen (IFDs) und die Datenpakete aus den Protokollen der jeweiligen IFD selektiert werden. Zur visuellen Unterstützung wird im unteren Bereich des Editors das selektierte Protokoll und seine Datenpakete grafisch dargestellt.
Die Mapping Functions werden, wie in der Abbildung zu sehen, im oberen Bereich, dem sogenannten Editorfenster, formuliert. Nach der Definition der Abbildungsvorschriften kann der Systemdesigner, durch den entsprechenden Mausklick auf den Button oder

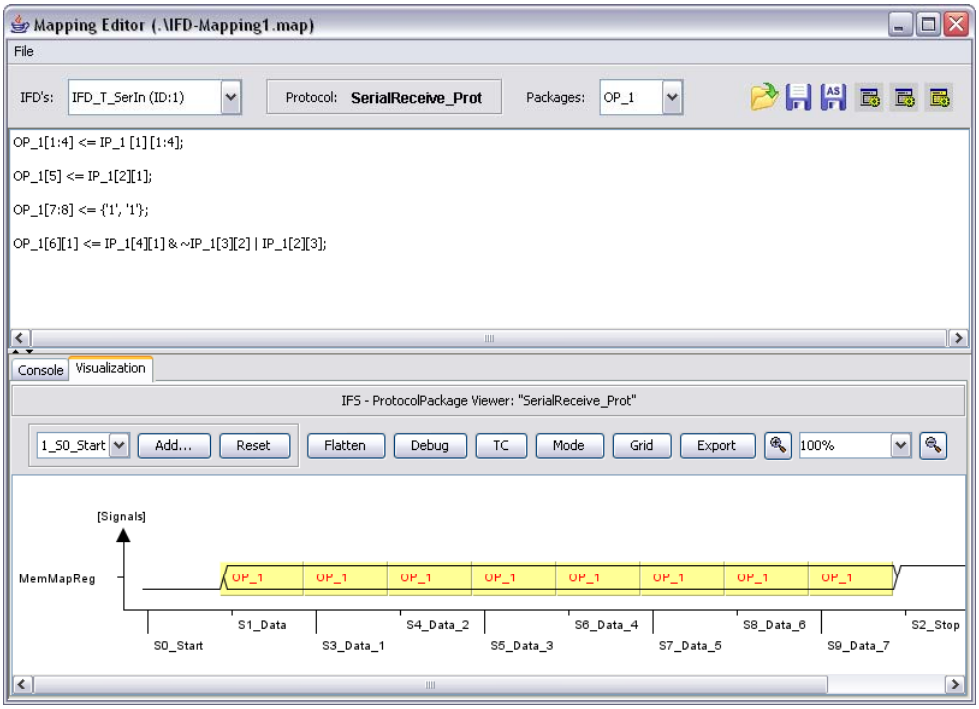

Abbildung 5.1: *Visualisierung im Mapping Editor*

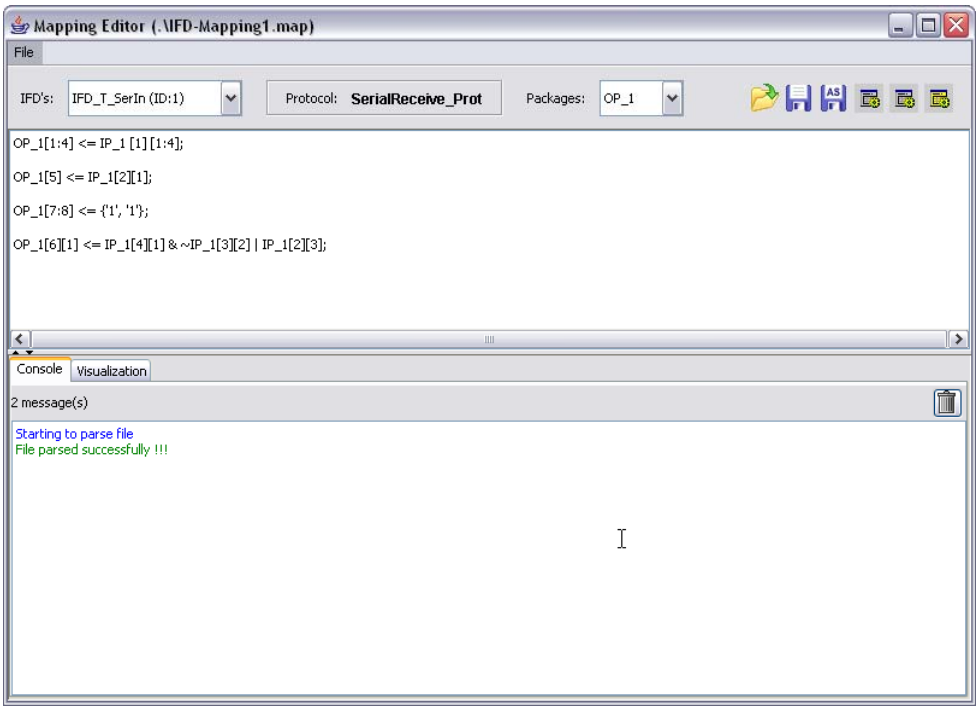

Abbildung 5.2: *Analyse der* Mapping Functions *im Mapping Editor*

einen Menüaufruf, die lexikalische und syntaktische Analyse durchführen (parsen). Das Ergebnis der Analyse wird in der sogenannten *Console*, im unteren Bereich der Editors, ausgegeben (siehe Abbildung 5.2). Durch einen weiteren Aufruf kann die semantische Analyse durchgeführt werden. War auch diese erfolgreich, so kann die Eingabe durch den "Accept"-Aufruf bestätigt werden, wonach der Mapping Editor geschlossen wird. Damit gilt die Definition der Mapping Functions als abgeschlossen. Die definierten Abbildungsregeln werden dabei in einer *.map* Datei gespeichert und können, falls erforderlich, jederzeit erneut bearbeitet werden.

Nach der Definition der Mapping Functions wird der Optimierungsprozess gestartet, dessen Ergebnisse in die weiteren Syntheseschritte einfliessen (vgl. Abbildung 4.1 auf Seite 41).

5.2 Datenstruktur

Die Klasse *IFD-Mapping* stellt mit ihren Methoden die wichtigste Java-Klasse bei der Umsetzung des IFD-Mapping Konzepts dar (siehe Abbildung 5.3). Konstruktor der Klasse erwartet als Parameter eine Liste aus Schnittstellenbeschreibungen. Während der Initialisierung werden aus den Protokollen der IFDs die Datenpakete und Frames generiert. Hierfür wird auf die Methoden der Klasse ProtocolPackageFactory zurückgegriffen. Über die entsprechenden *Getter*-Methoden der *IFD-Mapping* Klasse können die generierten Datenpakete und Frames dann abgefragt werden. So werden beispielsweisen im *Mapping Editor* diese Methoden benutzt, um die Auswahlbox mit den Datenpaketen zu füllen. Dabei können die Datenpakete und Frames jeweils nach deren "Richtung" gefiltert werden (*getDataInPackages(), getProtocolDataPackageOutFrames()*).
Die Methoden *parse()* und *interpret()* instanziieren einen *IFDMappingParser* und delegieren die Aufrufe an das Objekt weiter. Die Methode *parse()* führt die syntaktische Analyse durch, während die *interpret()*-Methode für die semantische Analyse verantwortlich ist. War das Parsen der Mapping Functions erfolgreich, so wird der generierte Ableitungsbaum (ParseTree) gespeichert. Einerseits dient er als Grundlage für die semantische Analyse, andererseits wird er auch zur Erzeugung einer IFB Instanz im zweiten Syntheseschritt benötigt.
Die Methoden *getMaxInternalBusWidthIn()* und *getMaxInternalBusWidthOut()* liefern anhand der Frames die maximal benötigte Datenbusbreite zwischen dem SH und PH. In der Abbildung 5.4 sind alle Methoden der *IFD-Mapping* Klasse dargestellt.

Die aus einem Protokoll generierten Datenpakete werden als Objekte der Klasse *ProtocolPackage* repräsentiert. Sie stellt eine Spezialisierung der Klasse *ProtocolMatrix* dar. Jedes *ProtocolPackage* Objekt besitzt einen eindeutigen Namen innerhalb des IFD-Mappings und darf – im Unterschied zur *ProtocolMatrix* – nur Elemente (*ProtocolMatrixNode*) mit dem *UseCase* "Data" beinhalten. Die Klasse *ProtocolPackage* bietet unter anderem Methoden zur Ergänzung der Datenpakete durch weitere Elemente: *extendRight(ProtocolMatrixNode packageNode), extendTop(ProtocolPackage protocolPackage).*

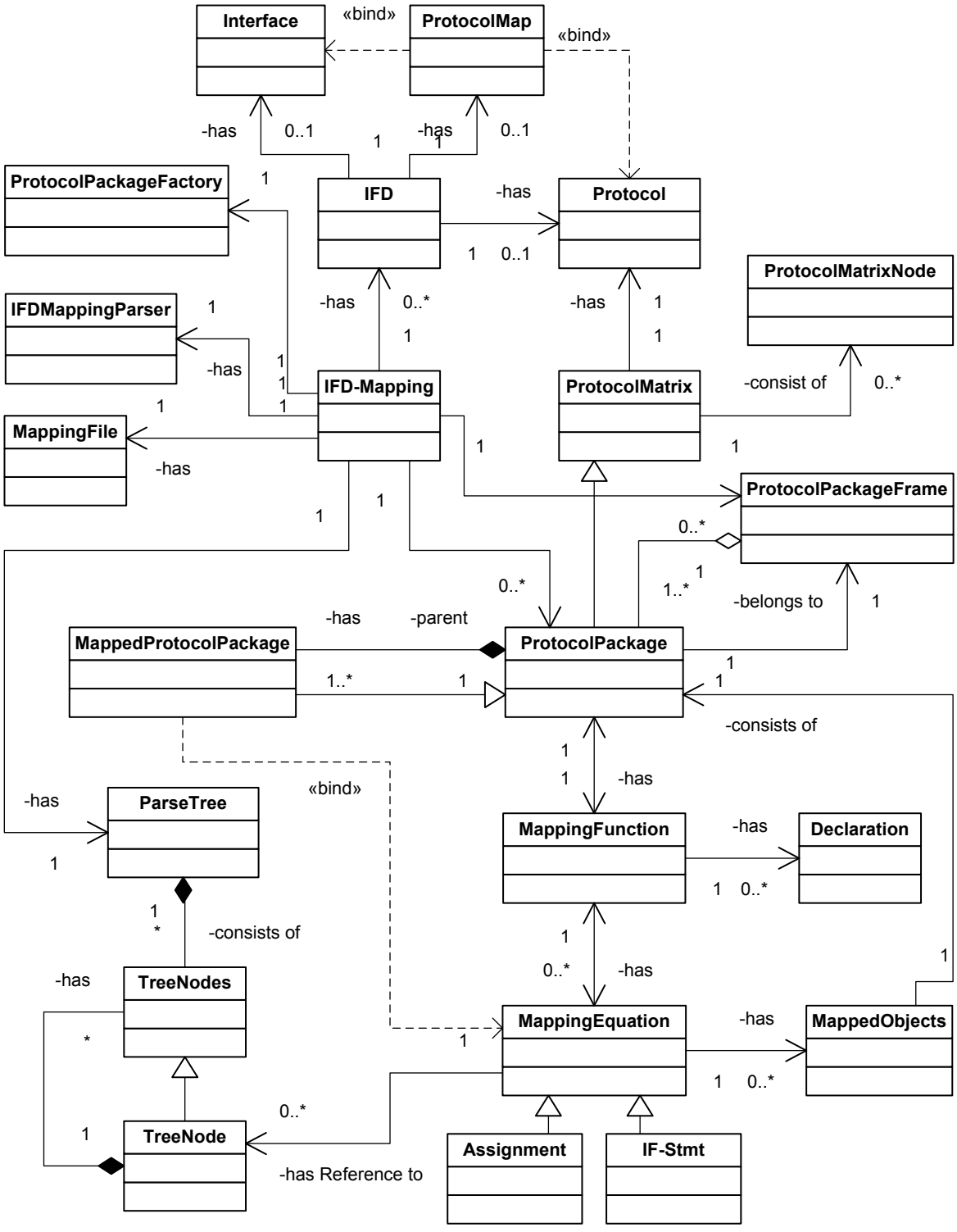

Abbildung 5.3: Klassendiagramm der IFB Synthesedddd

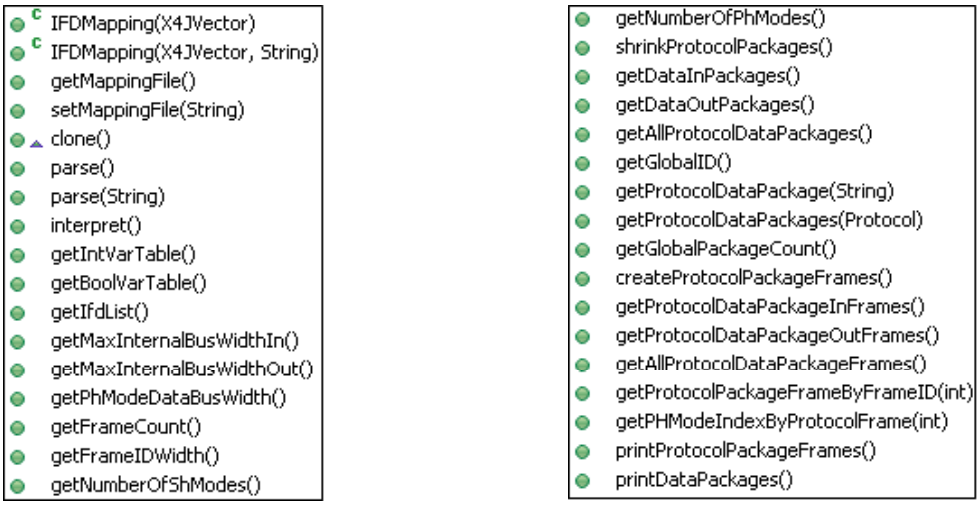

Abbildung 5.4: *Methoden der IFD-Mapping Klasse*

Jedes *ProtocolPackage* Objekt gehört zu einem Frame (*ProtocolPackageFrame*), und kann selbst wiederum aus mehreren Datenpaketen, den sogenannten *MappedProtocolPackage* Objekten, bestehen. Ein *ProtocolPackageFrame* Objekt besteht aus mindestens einem Datenpaket und bieten verschiedene Methoden um diese zu verwalten bzw. auszuwerten an (siehe Abbildung 5.5).

Eine Besonderheit stellt das Attribut *FrameId* dar. Neben der eindeutigen Kennzeichnung des Frames, besitzt die *FrameId* eine weitere wichtige Zusatzinformation, nämlich die Senderichtung. Gerade *FrameId's* weisen aus der Sicht des IFB auf einen eingehenden Frame, ungerade FrameId's auf einen ausgehenden Frame hin.
Die Klasse *MappedProtocolPackage* repräsentiert Datenpakete, die nach der Definition der Mapping Functions und anschliessenden Optimierung, generiert werden. Alle Elemente im *MappedProtocolPackage* wurden in den Abbildungsvorschriften (*MappingEquation*) einer MappingFunction "adressiert", und bilden somit die optimale (kleinste) Menge für die Datentransformation benötigten Datenbits. Die Erzeugung dieser Daten-

Abbildung 5.5: *Methoden der ProtocolPackageFrame Klasse*

pakete wird durch das "Schrumpfen" der Uhrsprungspakete (*ProtocolPackage*) erzielt, und von der *shrinkProtocolPackages()* Methode in der *IFD-Mapping* Klasse durchgeführt.

Jede Abbildungsvorschrift in einer Mapping Function wird als Objekt der Klasse *MappingEquation* repräsentiert. Eine *MappingEquation* kann dabei entweder eine einfache Zuweisung oder eine if-Anweisung sein. Instanzen dieser Klasse halten Referenzen auf alle in der Abbildungsvorschrift vorkommenden Datenpakete (*MappedObjects*), sowie auf den entsprechenden Knoten im Ableitungsbaum.

5.3 Lexikalischer und syntaktischer Analysierer

Die in Kapitel 3 vorgestellte Grammatik der Abbildungssprache in EBNF diente als Eingabe für den Generierungsprozess eines lexikalischen und syntaktischen Analysierers. Da JavaCC die Grammatik in einem proprietären Format benötig, musste diese entsprechend erstellt werden. Dabei werden die Nichtterminalen auf der linken Seite einer Produktionsregel in Form einer Java-Methode definiert. Die rechte Seite einer Produktionsregel, mit beliebigen Symbolen, wird dann im Rumpf dieser Methode notiert.

Die in Kapitel 3.4 bereits vorgestellte Produktionsregel in EBNF
"UnaryExpression = '~' UnaryExpression | '!' UnaryExpression | PrimaryExpression;"
sieht in JavaCC-Notation beispielsweise folgendermassen aus:

```
void UnaryExpression():
{}
{
  "~" UnaryExpression()
        |
  "!" UnaryExpression()
        |
  PrimaryExpression()
}
```

Damit während des Parsens ein Syntaxbaum erzeugt wird, wurde zusätzlich zu *JavaCC* das *JJTree* Tool verwendet. Hierfür musste die Grammatikdefinition nochmal erweitert werden. Ohne die Erweiterungen generiert *JJTree* für jede "Methode" eine Java-Klasse, die wiederum die Knoten in dem erzeugten Ableitungsbaum darstellen. Durch die sogenannten Labels in der JavaCC-Grammatik kann die Erzeugung einer Klasse verhindert, oder explizit erzwungen werden. Dabei kann zusätzlich durch die Parameter hinter den Labels die Knotenerzeugung gesteuert werden. Das vorherige Beispiel wurde daher modifiziert:

```
void UnaryExpression() #void:
{}
{
"~" UnaryExpression() #BitwiseComplNode(1)
      |
"!" UnaryExpression() #NotNode(1)
      |
      PrimaryExpression()
}
```

Durch das Label '#void' wird nun die Erzeugung eines *UnaryExpression* Knotens verhindert. Gleichzeitig wird aber explizit festgelegt, dass für jedes einzelne Terminalsymbol '~' oder '!' der Knoten *BitwiseComplNode* bzw. *NotNode* erzeugt werden soll.
Weiterführende Informationen zu JavaCC und JJTree befinden sich auf der offiziellen JavaCC-Homepage (siehe [7]).

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen dieser Studienarbeit wurde das Konzept des IFD-Mappings entwickelt und in den Interface Synthesis Design Flow integriert. Als Teil der szenariobasierten Eingabe für die Synthese der Schnittstellenadapter ermöglicht das IFD-Mapping die automatische Datentransformation zwischen inkompatiblen Protokollen. Anstelle einer sonst dafür üblichen Protokollsemantik, wurde im Rahmen des IFD-Mappings die Möglichkeit der Definition von Abbildungsvorschriften geschaffen.
Aufbauend auf dem Datenformat für Beschreibung von Schnittstellen (IFD) können nun Datenabbildungen als Mapping Functions definiert werden. Um die Abbildungsvorschriften formulieren zu können, wurde eine Sprache entwickelt und basierend auf einer in EBNF formulierten Grammatik spezifiziert. Die Sprache unterstützt folgende Operationen für die Definition der Abbildungsvorschriften:

- Zuweisung konstanter Werte
- Zuweisung eingehender auf ausgehende Bits (ohne diese zu verändern)
- Anwendung von booleschen Funktionen auf eingehende Bits
- Verwendung eines endlichen Zustandsautomaten (FSM) zur Datenmanipulation

Das Konzept des IFD-Mappings wurde durch seine Integration in den bestehenden Syntheseablauf des IFS-Editors realisiert. Hierfür wurde dafür die notwendigen Datenstrukturen entwickelt und ein *Mapping Editor* als GUI für die Definition der Mapping Functions in Java implementiert. Anhand der spezifizierten Grammatik der Abbildungssprache wurde ein Syntax- und Semantikanalysierer entwickelt, und in den IFS-Flow integriert. Weitere, in Java implementierte Methoden unterstützen die Synthese, indem sie die Informationen aus den Abbildungsvorschriften für die Parametrisierung und Erweiterung der Komponenten im IFB liefern. So wurden die SH-Modes im SH anhand der Abbildungsvorschriften aus dem IFD-Mapping implementiert, und Anpassungen für szenariobasierte Ausprägung der Protocol Handler und der Control Unit durchgeführt.

6.2 Ausblick

Eine mögliche Erweiterung des IFD-Mappings stellt die Umsetzung des in Kapitel 3 vorgestellten DFS Algorithmuses im Rahmen der Semantikanalyse. Die Prüfung auf die

Kreisfreiheit in Datenpaketgraphen stellt eine wichtige und notwendige Voraussetzung für die Erfüllung des EVA Schemas und der konfliktfreien Protokollausführung.

Der Weiteren wäre eine Erweiterung der Sprache durch die Konstrukte für die Wiederverwendung. Wie in höheren Programmiersprachen, wäre die Wiederverwendung durch Komposition oder Vererbung denkbar. So ließen sich beispielsweise während der Definition der Mapping Functions bestimmte Methoden (z.B. CRC Prüfung) aus anderen Mapping Functions aufrufen oder deren Eigenschaften vollständig erben.

A Anhang

A.1 Grammatik in EBNF

```
MappingFunction          = {ImportDeclaration} {MappingDeclaration};
ImportDeclaration        = 'import' Identifier;
Identifier               = Letter {Letter | Digit};
Letter                   = 'A' | 'B' | 'C' | ··· | 'Z' | 'a' | 'b' | 'c' | ··· | 'z';
Digit                    = '0' | '1' | '3' | ··· | '9';
MappingDeclaration       = VariableDeclarator ';' | Statement;
VariableDeclarator       = VariableType Identifier ['=' ConditionalOrExpression];
VariableType             = ('boolean' | 'int' | 'bit') ['[]'] ['[]'];
Assignment               = PackageOrVariable AssignmentOperator
                           ConditionalOrExpression;
AssignmentOperator       = '<=' | '=';
ConditionalOrExpression  = ConditionalAndExpression {'||' ConditionalAndExpression};
ConditionalAndExpression= InclusiveOrExpression {'&&' InclusiveOrExpression};
InclusiveOrExpression    = ExclusiveOrExpression {'|' ExclusiveOrExpression};
ExclusiveOrExpression    = AndExpression {'^' AndExpression};
AndExpression            = EqualityExpression {'&' EqualityExpression};
EqualityExpression       = AdditiveExpression {('==' '!=') AdditiveExpression};
AdditiveExpression       = UnaryExpression {'+' UnaryExpression};
UnaryExpression          = '~' UnaryExpression | '!' UnaryExpression |
                           | PrimaryExpression;
PrimaryExpression        = PackageOrVariable | Literal | ConstantArray |
                           | '{' ConstantArray {',' ConstantArray} '}' |
                           | '(' ConditionalOrExpression ')';
PackageOrVariable        = ProtocolPackage ['(' IntegerLiteral ')']
                           [ProtocolPackageSuffix] | Identifier;
ProtocolPackage          = ('IP_' | 'OP_') IntegerLiteral;
ProtocolPackageSuffix    = '[' IntInterval ']' [ '[' IntInterval ']' ];
Literal                  = IntegerLiteral | BooleanLiteral | BitLiteral;
ConstantArray            = '{' Literal {',' Literal} '}';
IntInterval              = IntegerLiteral [':' IntegerLiteral];
IntegerLiteral           = (Digit - '0') {Digit};
BooleanLiteral           = 'true' | 'false';
BitLiteral               = "'('1' | '0')'";
```

```
Statement          = Block | IfStatement | Assignment ';' ;
Block              = '{' {MappingDeclaration}' }';
BlockStatement     = VariableDeclarator ';' | Statement;
IfStatement        = 'if' '('ConditionalOrExpression')' Statement
                     ['else' Statement];
```

Literaturverzeichnis

[1] Janaki Akella and Kenneth L. McMillan. Synthesizing Converters Between Finite State Protocols. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 410–413, Washington, DC, USA, 1991. IEEE Computer Society.

[2] Dieter Averberg. Synthese von deadline-konformen Protokollkonvertern für heterogene verteilte Anwendungen. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, März 2005.

[3] Christian Behler. Datenflussoptimierung in rekonfigurierbarer Hardware durch Pipelining. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, Mai 2005.

[4] Gilles Bertrand Gnokam Defo. VHDL Codegenerierung für rekonfigurierbare Schnittstellen in eingebetteten Systemen. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, Mai 2005.

[5] Stefan Ihmor. Entwurf von Echtzeitschnittstellen am Beispiel interagierender Roboter. Master's thesis, Universität Paderborn, Warburger Str. 100, 33098 Paderborn, November 2001.

[6] Stefan Ihmor, Nilson Bastos Jr., Rafael Cardoso Klein, Markus Visarius, and Wolfram Hardt. Rapid Prototyping of Realtime Communication – A Case Study: Interacting Robots. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, pages 186 – 192, San Diego, CA, June.

[7] JavaCC. *Java Compiler Compiler (JavaCC^{TM}) - The Java Parser Generator.* https://javacc.dev.java.net/.

[8] Tobias Loke. Synthese von Kommunikationsstrukturen in verteilten eingebetteten Systemen. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Fürstenallee, 33102 Paderborn, März 2005.

[9] Roberto Passerone. Automatic Synthesis of Interfaces Between Incompatible Protocols.

[10] Uwe Schöning. *Theoretische Informatik - Kurzgefaßt.* Spektrum Akademischer Verlag, 1997.

[11] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms.* The MIT Press Cambridge, Massachusetts London, England, 1998.

[12] Ingo Wagener. *Theoretische Informatik - eine algorithmische Einführung.* B. G.Teubner Stuttgart * Leipzig, 1999.

[13] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, 1996.