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

Modellbasierte Spezifikation von Schnittstellen im Hardware Entwurf

Diplomarbeit

Studiengang Ingenieurinformatik
Schwerpunkt Informatik
Vertiefungsrichtung Elektrotechnik

von

Michel Carmel Kouamo Sime

Husenerstraße 46

33098 Paderborn

betreut durch

Dipl.-Inform. Stefan Ihmor

vorgelegt bei

Prof. Dr. rer. nat. Franz Josef Rammig

Prof. Dr. habil. Wolfram Hardt, TU Chemnitz

im

Februar 2005

Dank und Erklärung

Dieses Dokument entstand im Rahmen einer Diplomarbeit in der Arbeitsgruppe von Prof. Dr. rer. nat. Franz Josef Rammig (HNI) der Universität Paderborn. Während der Anfertigung der Diplomarbeit lernte ich Problemstellungen der Informatik zielstrebig zu studieren.

Für das interessante Diplomarbeitsthema möchte ich mich bei Franz J. Rammig und Wolfram Hardt bedanken. Besonderer Dank gilt Stefan Ihmor, der mich über den Zeitraum der Erstellung der Diplomarbeit fachlich engagiert betreut hat. Auch Martin Kardos und Andreas Scholand bin ich für ihre fachliche Unterstützung zu Dank verpflichtet. Nicht zuletzt möchte ich Andrea Kwamo-Kamdem, Oliver Fick, Andreas Scholand und Martin Kardos für das intensive Korrekturlesen dieser Arbeit danken.

Meinen Eltern, die mir das Studium ermöglicht und mich stets unterstützt haben, möchte ich ebenfalls Dank sagen.

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, im Februar 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

1.1. Motivation

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

1.2. Problemstellung

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

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

1.3. Aufgabenstellung

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

1.4. Gliederung der Arbeit

Diese Arbeit ist in sechs Kapitel gegliedert.

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

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

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

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

Kapitel sechs fasst die Ergebnisse zusammen und gibt einen Ausblick.

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

2. Stand der Technik

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

2.1. UML2.0

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

2.1.1. Warum eine neue Version?

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

2.1.2. Was ist neu an UML2.0

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

- Interaktionsübersichtsdiagramm
- Kompositionsstrukturdiagramm
- Timing-Diagramm
- Kommunikationsdiagramme

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

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

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

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

2.1.3. Verwendete Diagrammtypen

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

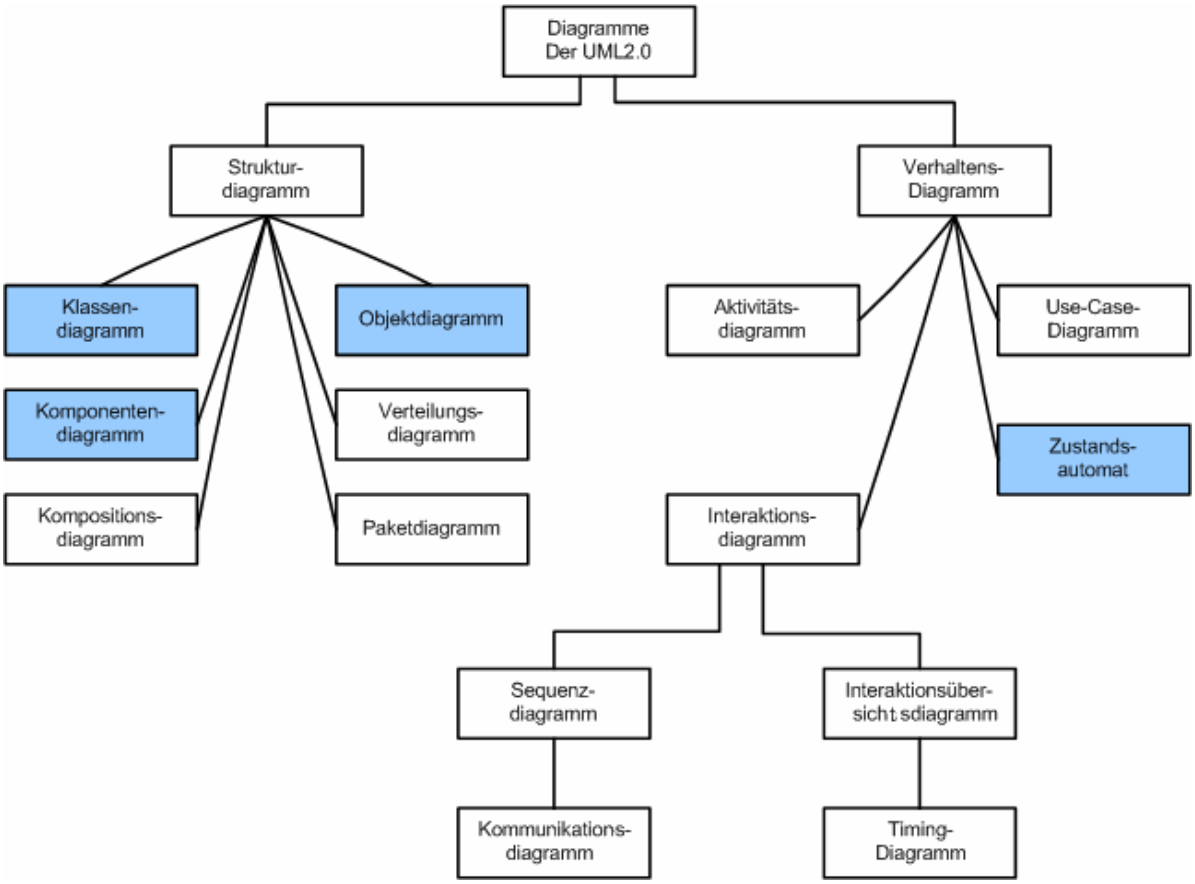


Abbildung 2.1.: Diagrammtypen der UML2.0 [Jec04b]

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

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

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

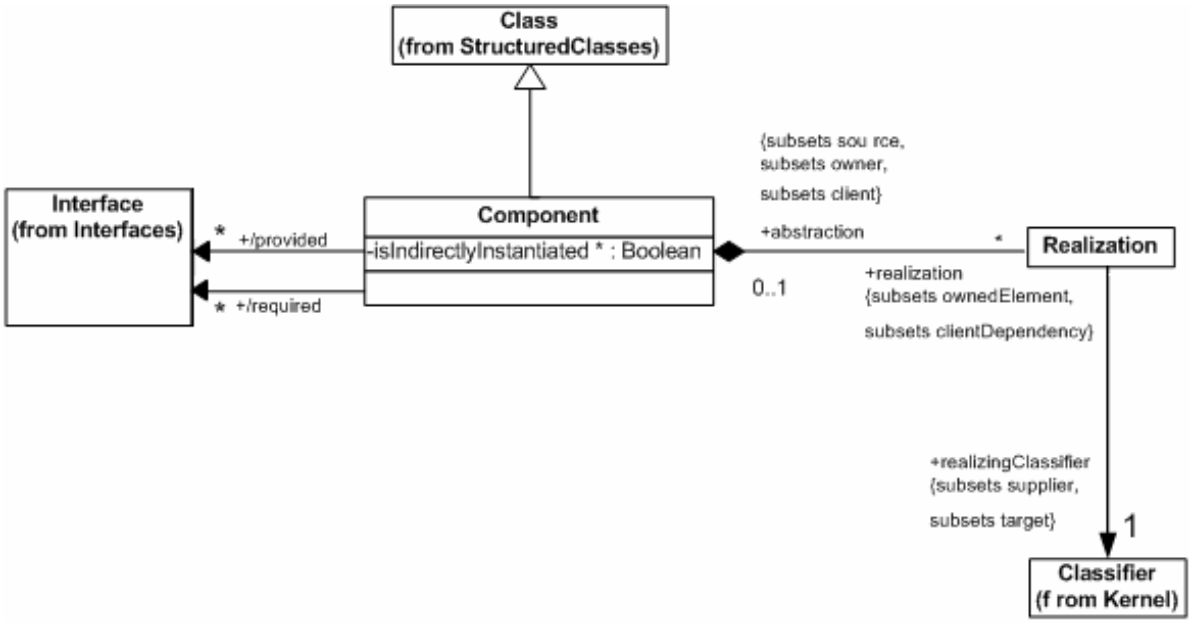


Abbildung 2.2.: Grundlegende Basis der Komponentenstruktur [Uni04]

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

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

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

Schnittstellen

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

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

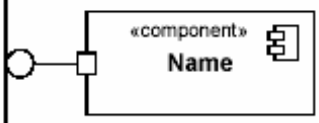
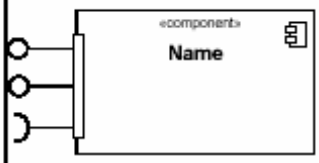
Node Type	*Notation*	*Reference*
Component has provided Port (typed by Interface)		See “Port”.
Component uses Interface		See “Interface”.
Component has required Port (typed by Interface)		See “Port”.
Component has complex Port (typed by provided and required Interfaces)		See “Port”.

Tabelle 2.1.: Verfügbare UML Schnittstellen [Uni04]

- Implementierte Schnittstelle *(provided interface)*

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

- Benötigte Schnittstelle *(required interface)*

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

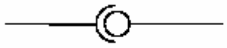
Path Type	Notation	Reference
Assembly connector		See "assembly connector". Also used as notation option for wiring between interfaces using Dependencies.

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

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

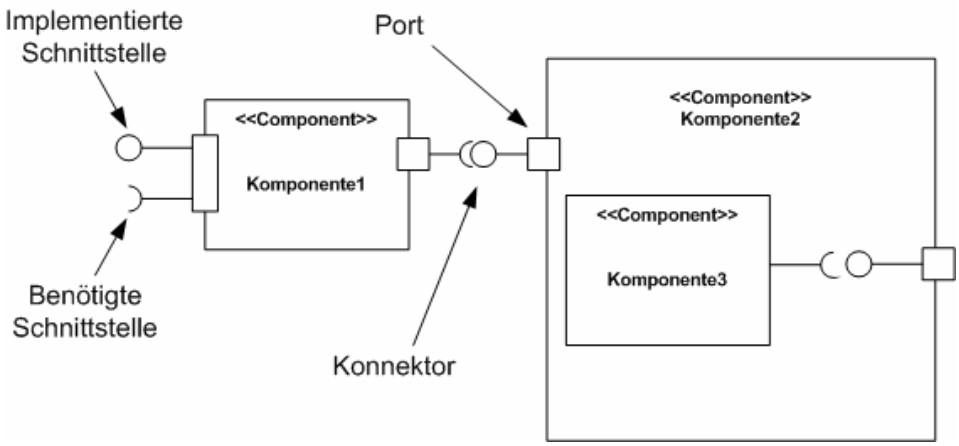


Abbildung 2.3.: Notation eines Ports mit Schnittstelle [Jec04b]

Port

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

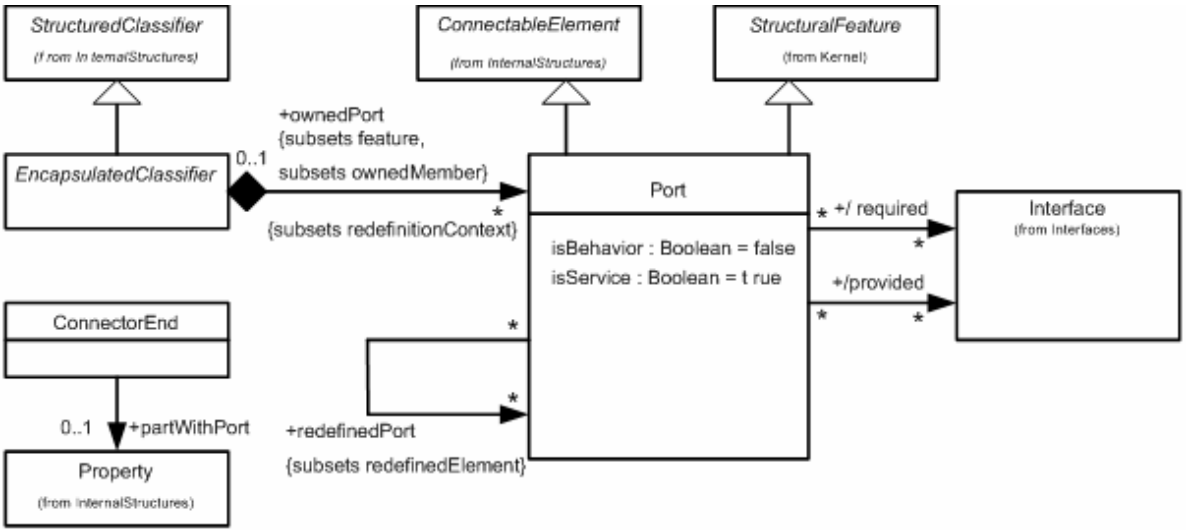


Abbildung 2.4.: Port Metamodell [Uni04]

- Zustandsautomat

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

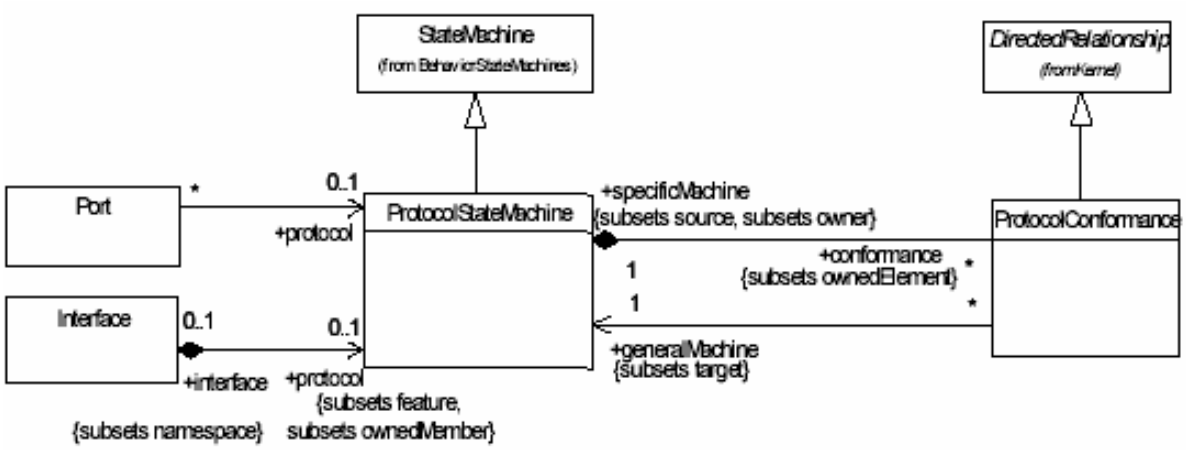


Abbildung 2.5.: Protokoll Zustandsmaschine [Uni04]

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

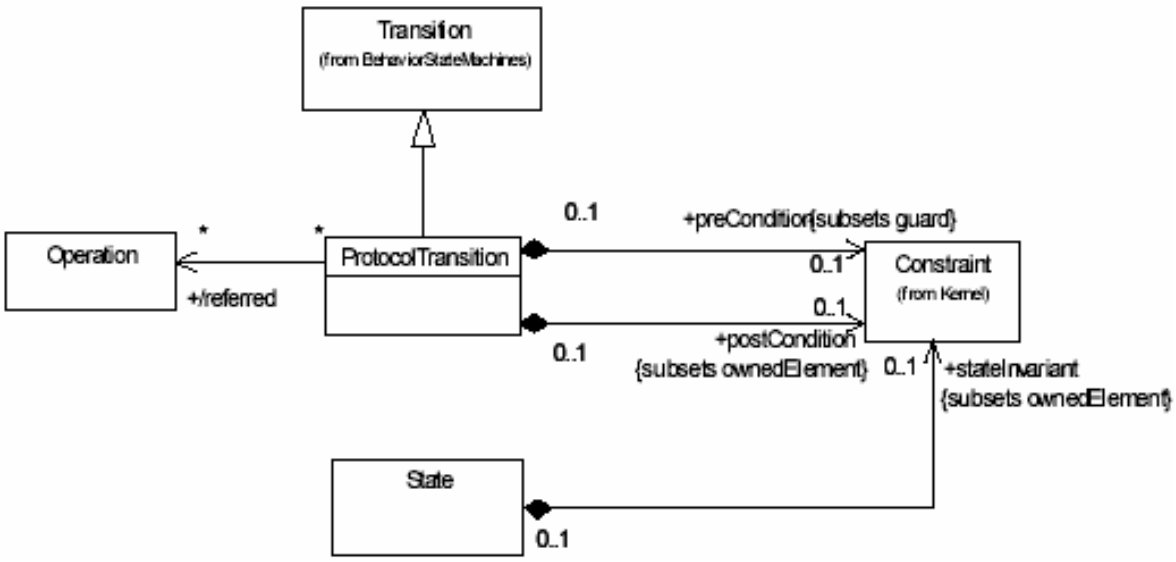


Abbildung 2.6.: Constraint Klasse [Uni04]

2.1.4. Erweiterungsmechanismen

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

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

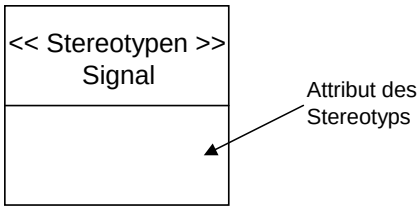


Abbildung 2.7.: Definition des Stereotyps

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

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

2.2. Das IFS-Format

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

2.2.1. Einführung in Interface Synthese

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

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

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

2.2.2. Systemarchitektur

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

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

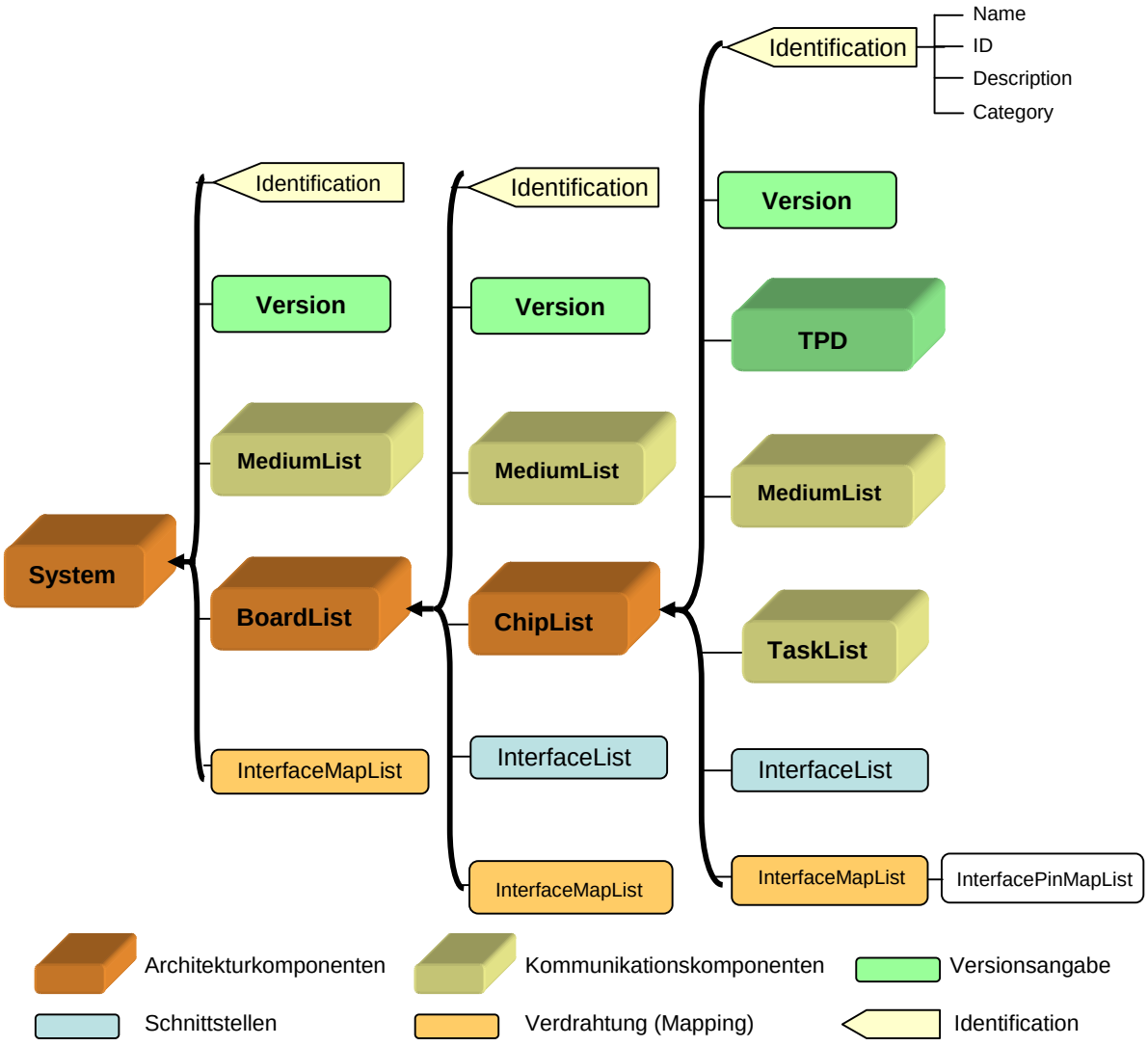


Abbildung 2.8.: Architektur- und Kommunikationskomponenten

Architekturkomponenten

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

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

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

Kommunikationskomponenten

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

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

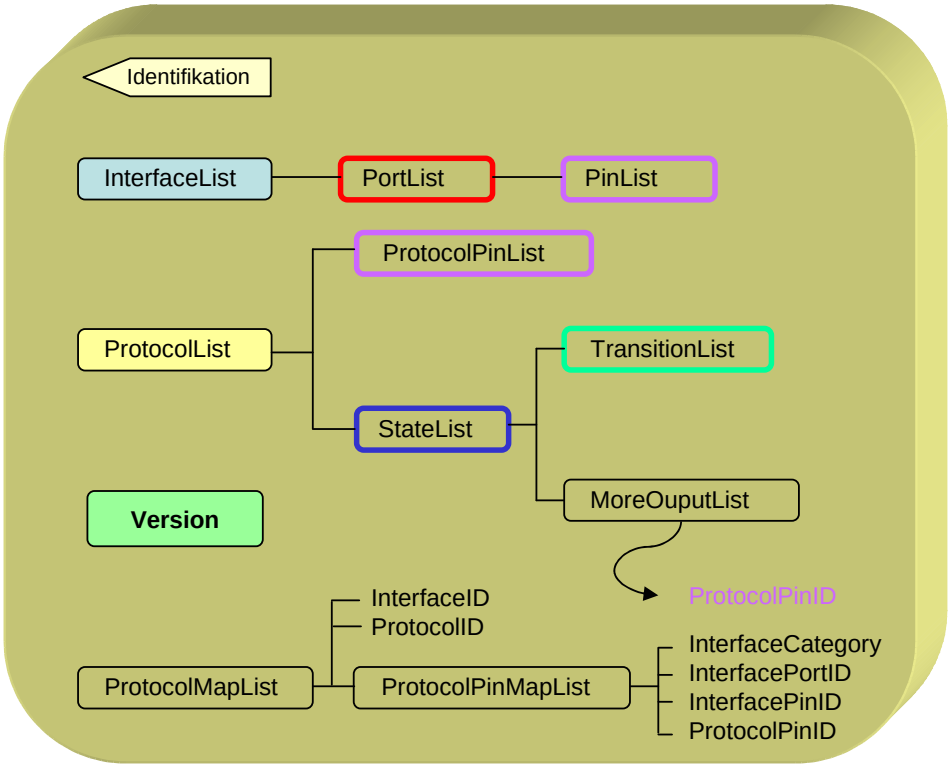


Abbildung 2.9.: Task im IFS-Konzept

2.2.3. IFS-Editor

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

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

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

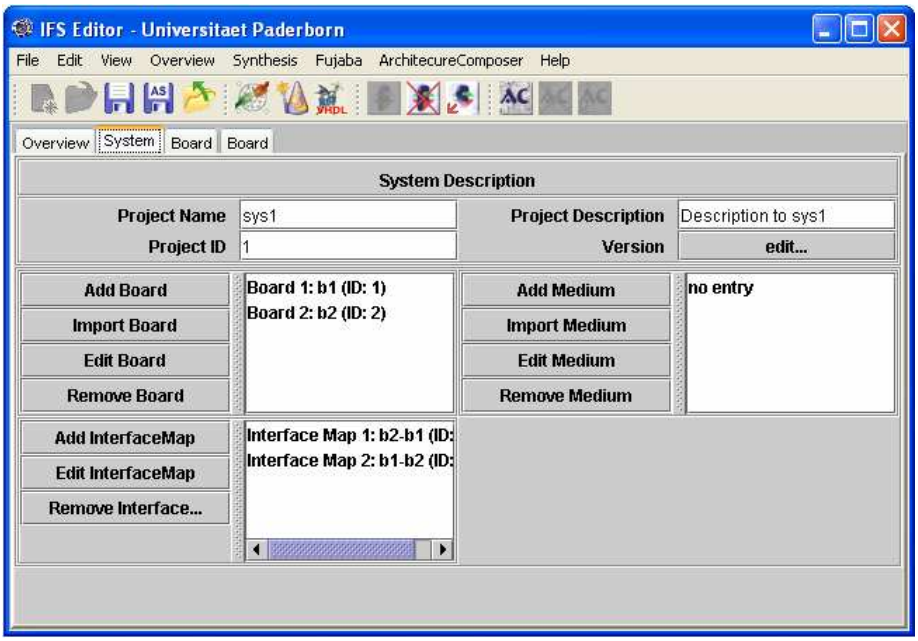


Abbildung 2.10.: IFS-Editor

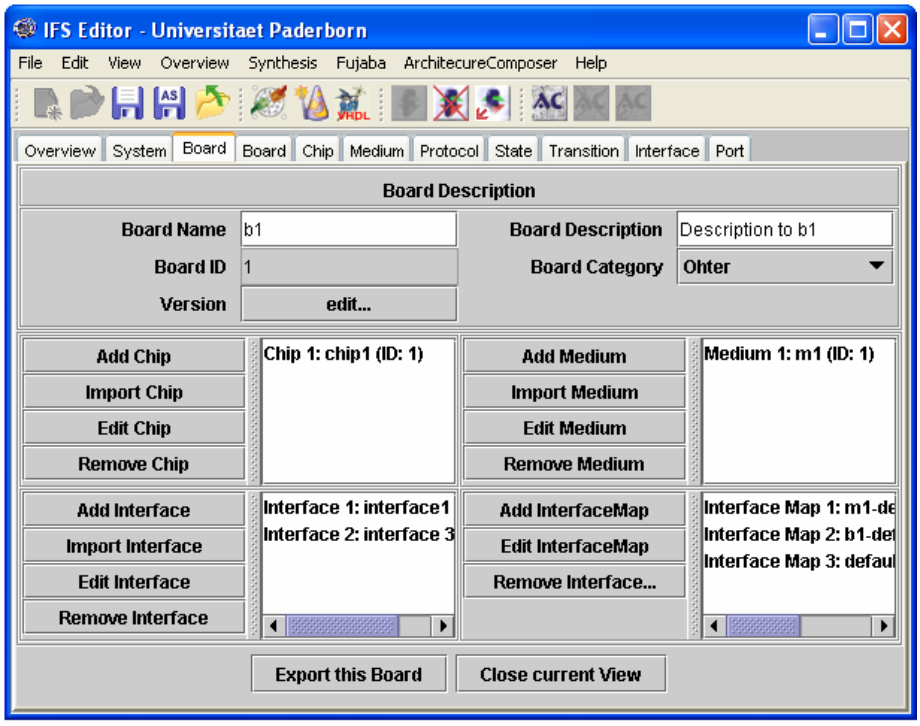


Abbildung 2.11.: IFS-Editor Board-Sicht

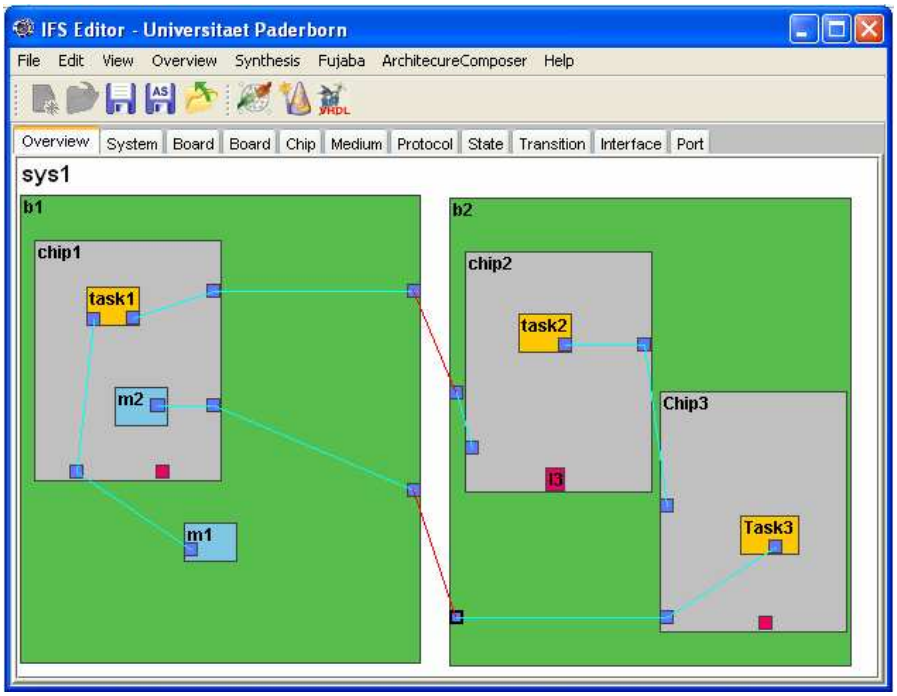


Abbildung 2.12.: IFS-Editor Overview

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

2.3. Das WerkZeug Fujaba

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

2.3.1. Warum Fujaba?

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

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

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

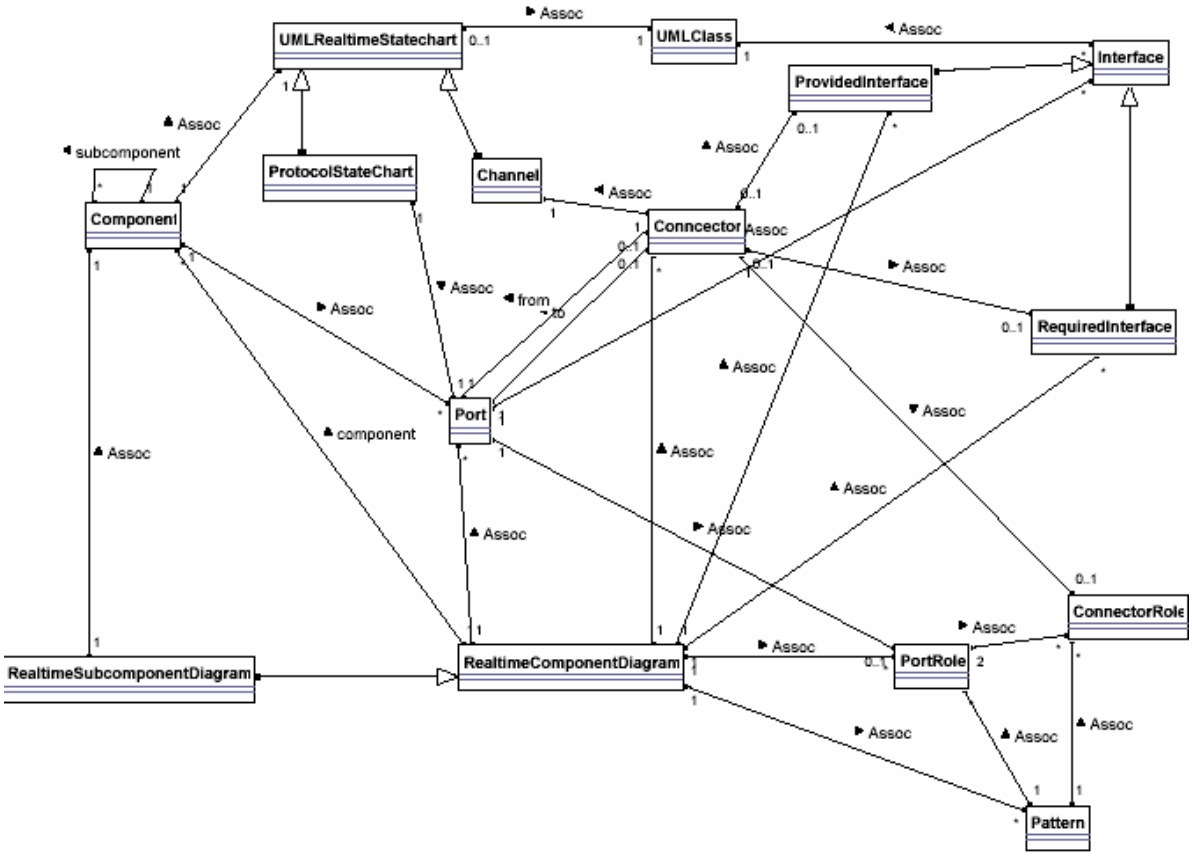


Abbildung 2.13.: FUJABA-Metamodell des RT-Component Diagramms

3. Lösungskonzepte

3.1. Erweiterung des bestehenden Modellierungskonzepts

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

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

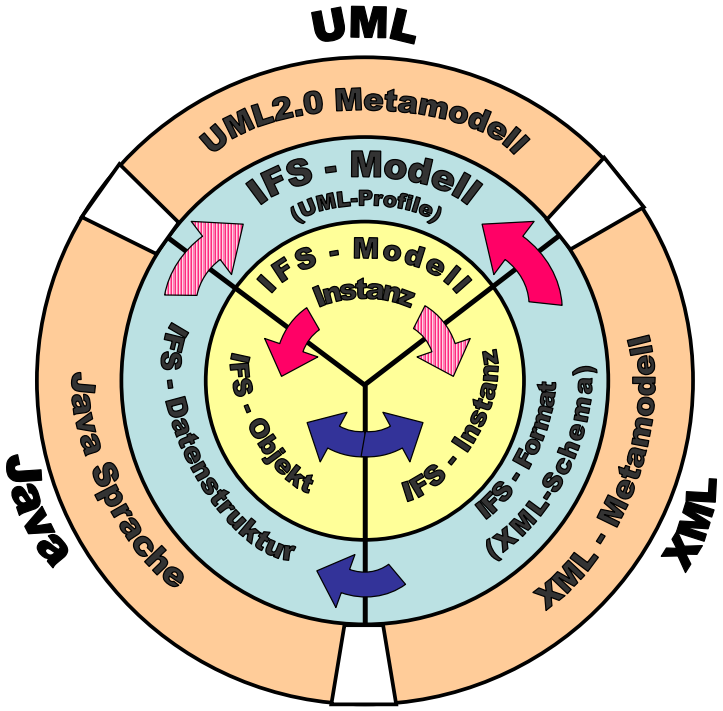


Abbildung 3.1.: Das um UML2.0 erweiterte Modellierungskonzept

3.2. Das Modellierungskonzept in UML2.0

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

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

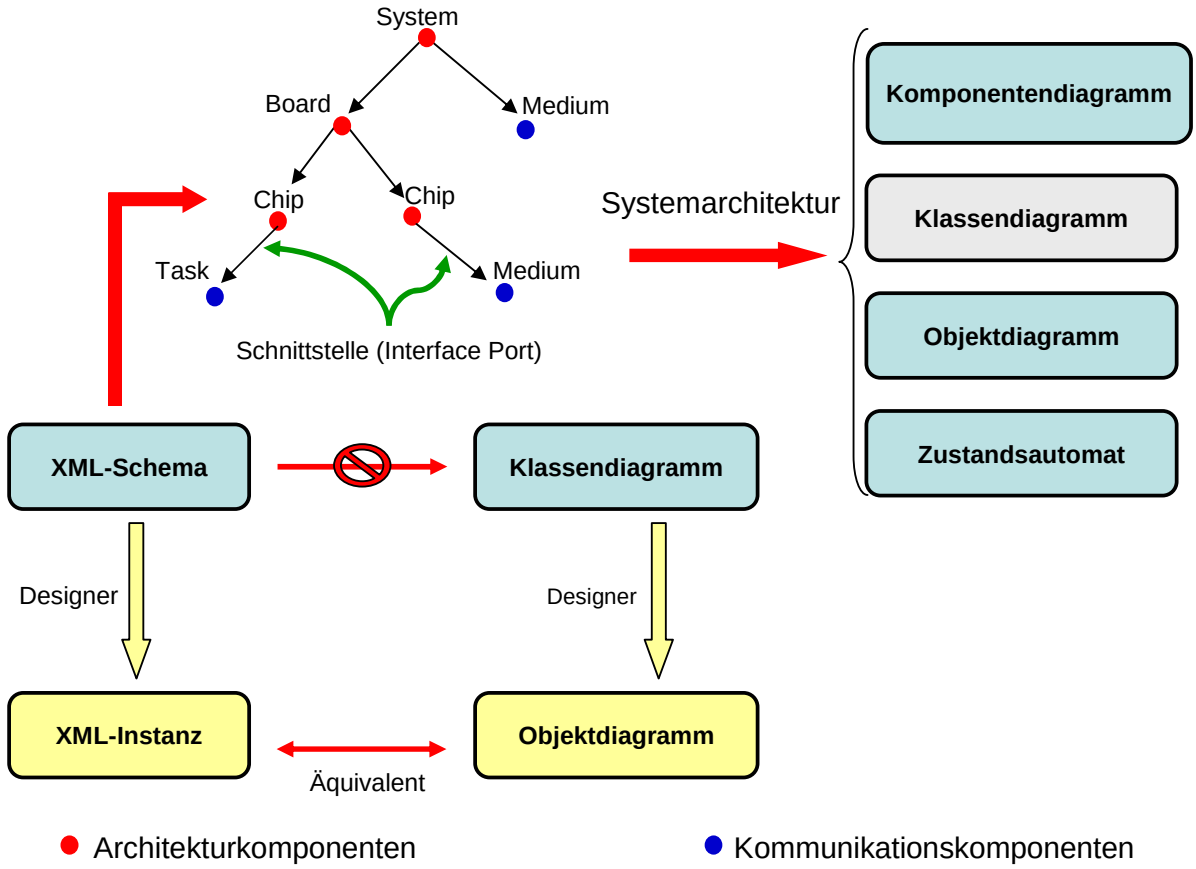


Abbildung 3.2.: Das Modellierungskonzept in UML2.0

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

3.3. Modelltransformation und Lösungsansätze

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

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

3.3.1. Modelltransformation

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

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

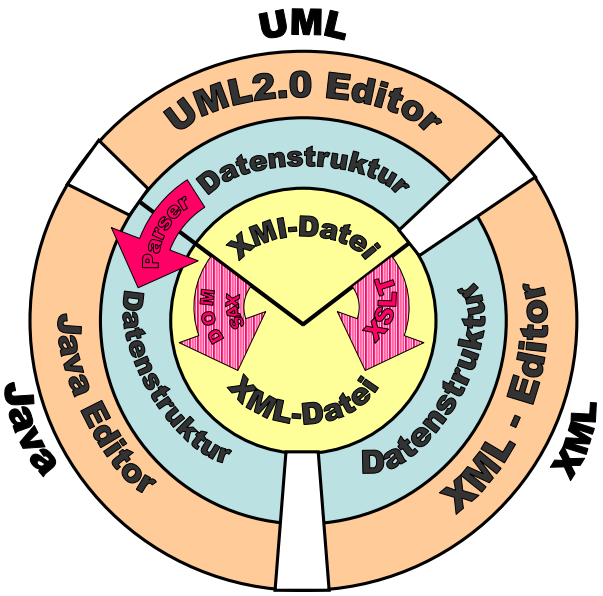


Abbildung 3.3.: Modelltransformation

3.3.2. Lösungsansätze

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

XML Parser

- **SAX [Bre01]**
  SAX (Simple API for XML) ist eine standardisierte Möglichkeit um XML Dateien zu bearbeiten. Das Programm kann allerdings nicht in einem XML Dokument hin- und herspringen und kann nur in einem beschränkten Anwendungsgebiet eingesetzt werden. SAX ist sequentiell. Die Informationen über ein XML Dokument stehen nur in dem Moment des Parsens zur Verfügung und gehen verloren, sobald der Parser fortschreitet. Mit einer selbst geschriebenen Parser-Callback-Methode können Daten, Elemente, Attribute und Strukturen im zu parsenden Dokument manipuliert werden. Ein Vorteil von SAX ist, dass beim Parsen nur ein kleiner Teil der XML Datei im Speicher sein muss und sich damit die Geschwindigkeit erhöht. Dies ist aber ein Nachteil, wenn man mehrere Informationen in der ganzen Datei für die Bearbeitung benötigt, da man selbst entsprechende Datenstrukturen zur Erfassung und Erhaltung der Daten im Speicher entwickeln muss. Aus diesem Grund ist es sehr aufwendig und nicht zweckmäßig diesen Ansatz zum Exportierten von UML Modellen das IFS-Schemas zu verwenden. Ein Beispielcode befindet sich im Anhang unter A.2.1

- **DOM [Bre01]**
  DOM (Document Object Model) ist die zweite Variante, um XML Dateien auszuwerten. DOM ist eine in Java implementiertes Konzept mit definierten Methoden, die das Lesen, Anlegen und Modifizieren von Objekten innerhalb von Dokumenten ermöglichen. Sie stellt im Gegensatz zu SAX eine Baum-Repräsentation der Objekte zur Verfügung. Dies geschieht, indem die gesamte XML Datei gelesen wird und benötig daher viel Speicherplatz. Der Vorteil bei diesem Ansatz ist, dass alle Elemente der XML Datei in einer hierarchischen Struktur vorliegen und Zugriffe darauf mit Hilfe der objektorientierten Datenstruktur erfolgen. Auch dieser Ansatz wurde in dieser Arbeit nicht weiter verfolgt, da es neben dem hohen Speicherbedarf auch recht aufwendig ist DOM Dokumente zu tranversieren. Hinzu kommt, dass hier XML bzw. XMI Dateien geladen (DOM) und gespeichert (UML-Tool und DOM) werden müssen. Das Beispiel in Anhang unter A.2.2 zeigt, wie ein DOM-Parser-Programm geschrieben werden kann, um XML Dokumente einzulesen.

- **XSLT [Bre01]**
  Einen weiteren Ansatz bietet XSLT (Extensible Stylesheet Language Transformation). XSLT ist eine Programmiersprache zur Transformation von XML-Dokumenten. Sie erlaubt die Definition von Umwandlungsregeln von einem XML Quellformat auf beliebige andere Zielformate. Nachdem die Stylesheet-Regeln für die Transformation definiert wurden, ist jeder XSLT-Prozessor in der Lage, den XSLT-Stylesheet zu lesen und eine XML-Datei in das gewünschte Ausgabeformat zu transformieren. Als Erläuterung wurde mit

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

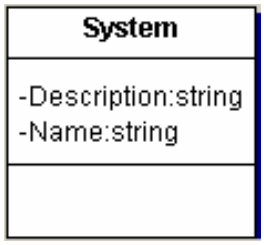


Abbildung 3.4.: Die Klasse “System”

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

Beispiel 3.1: XML-Beispieldatei

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

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

Beispiel 3.2: Das Sytlesheet zu Beispiel 3.1

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

Beispiel 3.3: Die resultierende IFS-Instanz

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

Parsen der Datenstruktur im Speicher

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

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

4. Modellierung

4.1. Grundgedanken

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

4.2. Aufbau des Modells

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

Identifikation

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

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

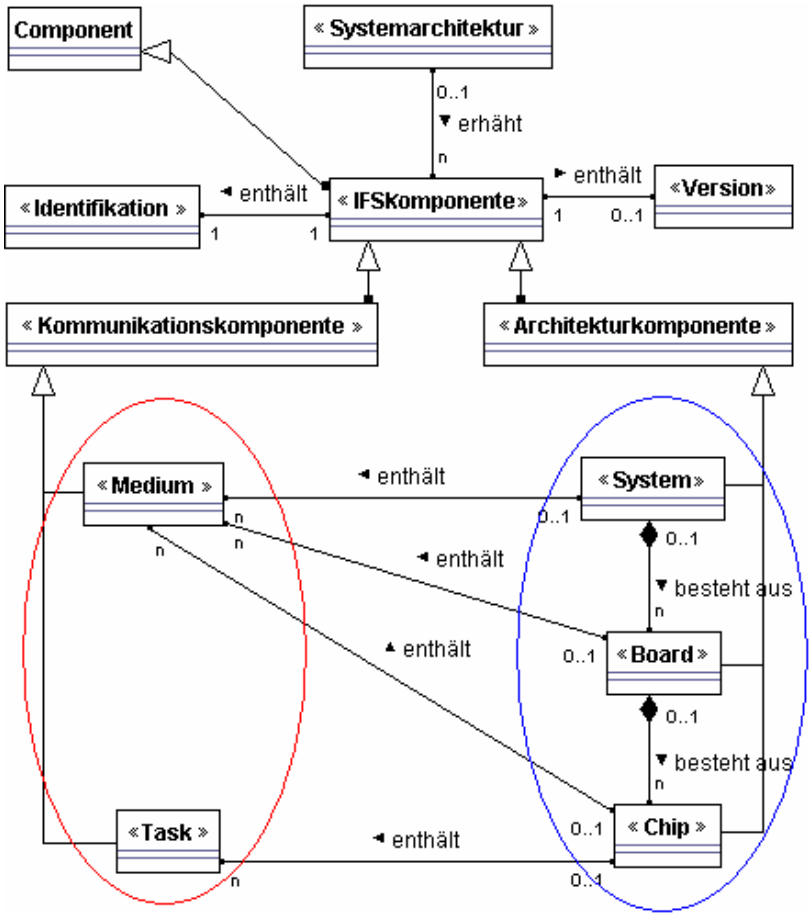


Abbildung 4.1.: Klassendiagramm der Systemarchitektur

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

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

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

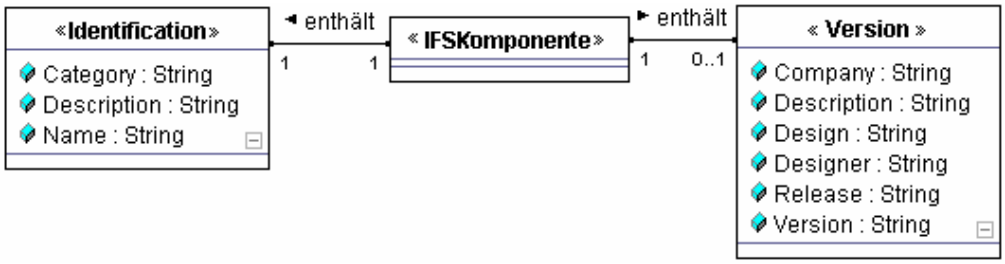


Abbildung 4.2.: Die Klasse “IFSKomponente”

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

4.2.1. Strukturbeschreibung

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

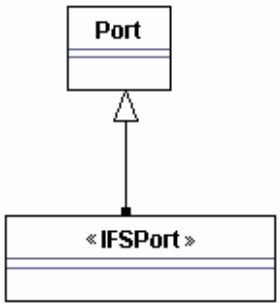


Abbildung 4.3.: Stereotype “IFSPort”

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

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

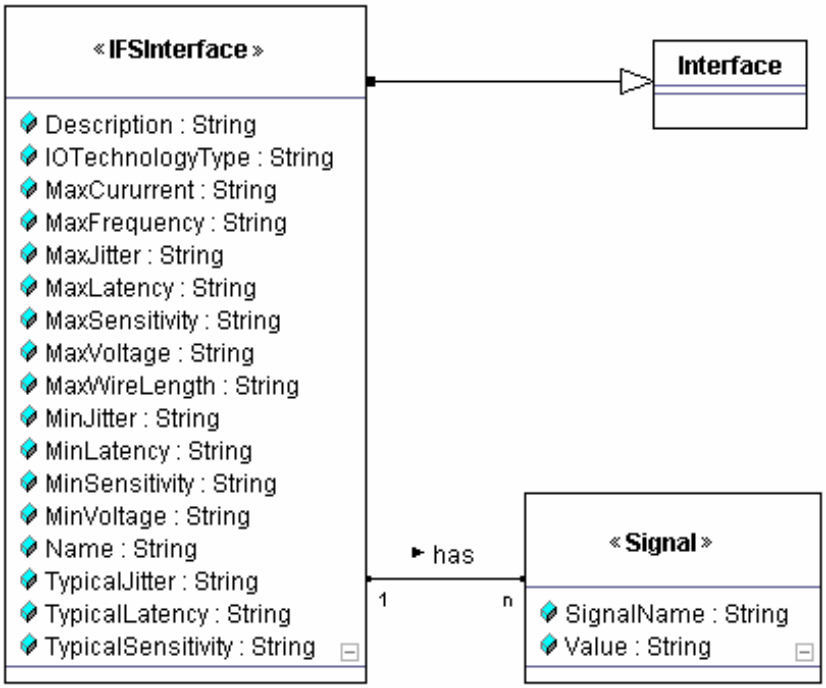


Abbildung 4.4.: Die Klasse “Interface”

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

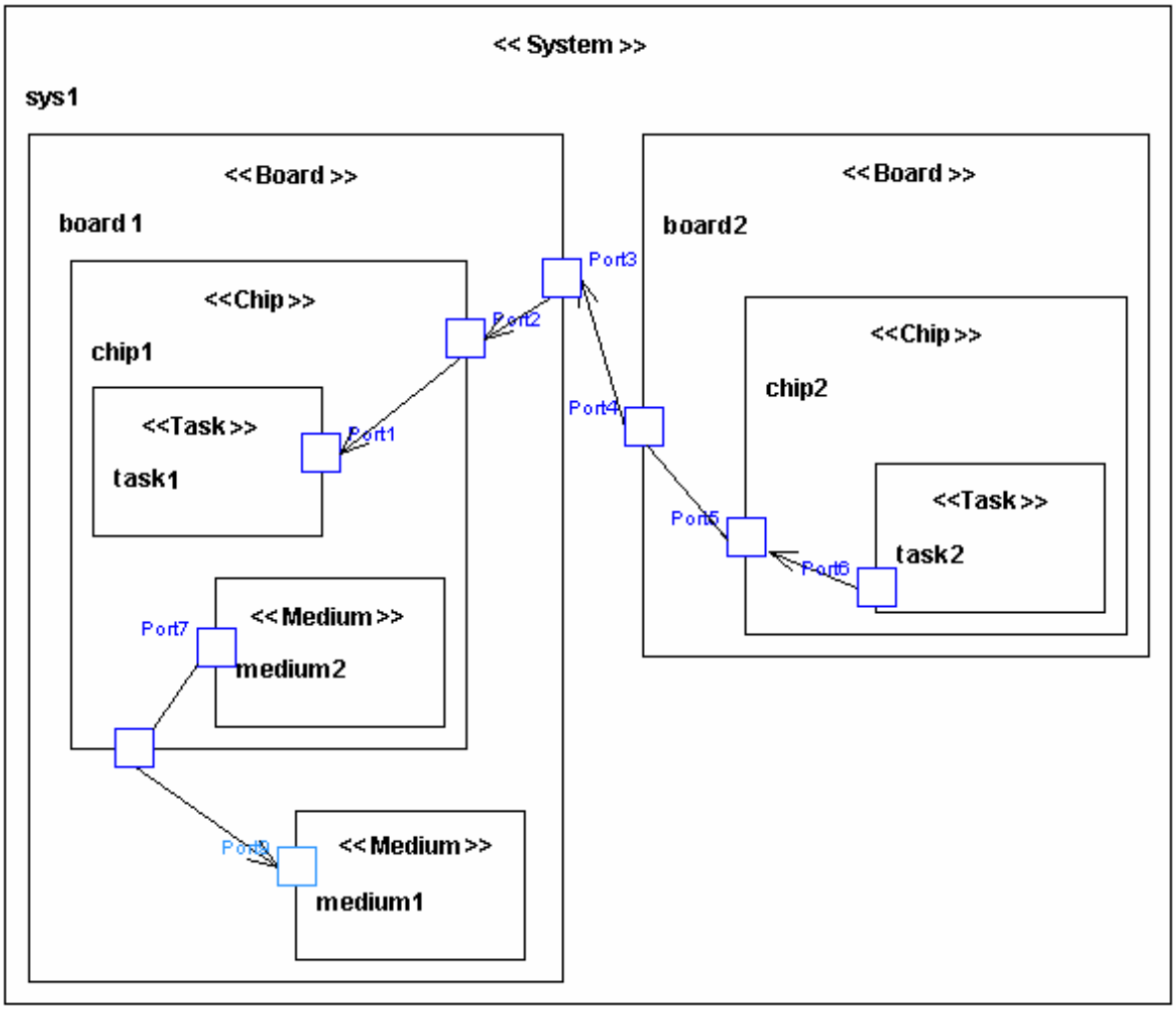


Abbildung 4.5.: Die Systemarchitektur als Komponentendiagramm

4.2.2. Verhaltensbeschreibung

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

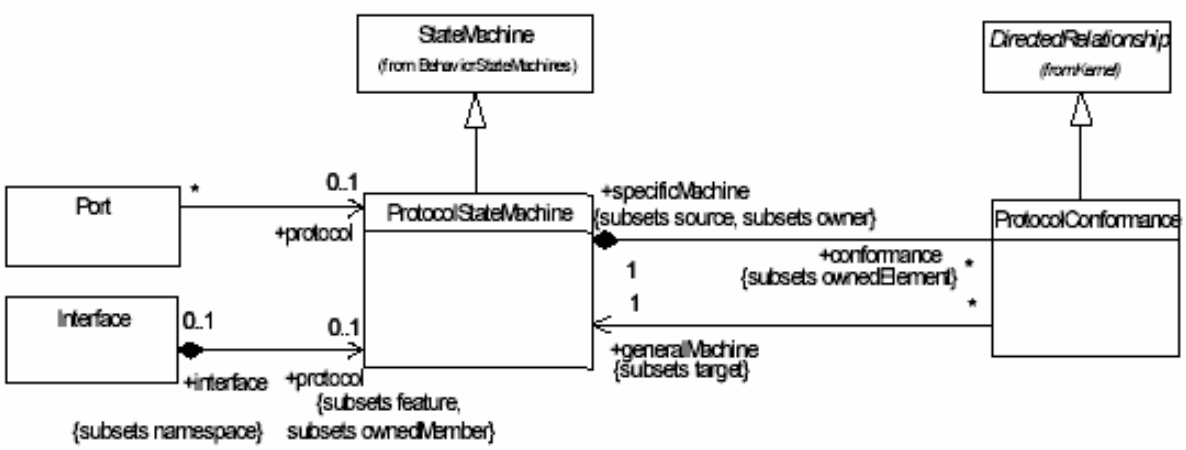


Abbildung 4.6.: UML Protocol state machines [Uni04]

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

ProtocoloPins

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

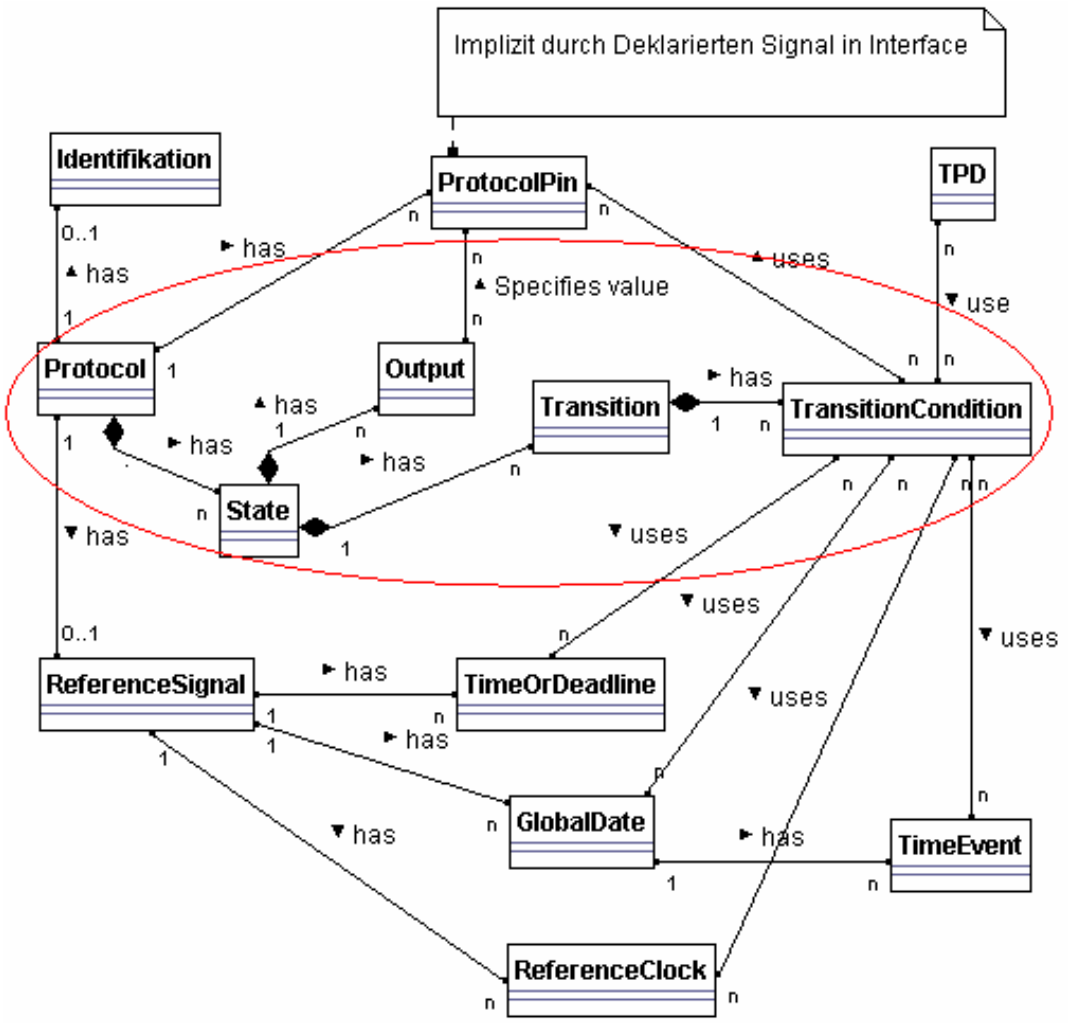


Abbildung 4.7.: Aufbau des Kommunikationsprotokolls des IFS-Schemas als Klassendiagramm

Der Zustandsautomat

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

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

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

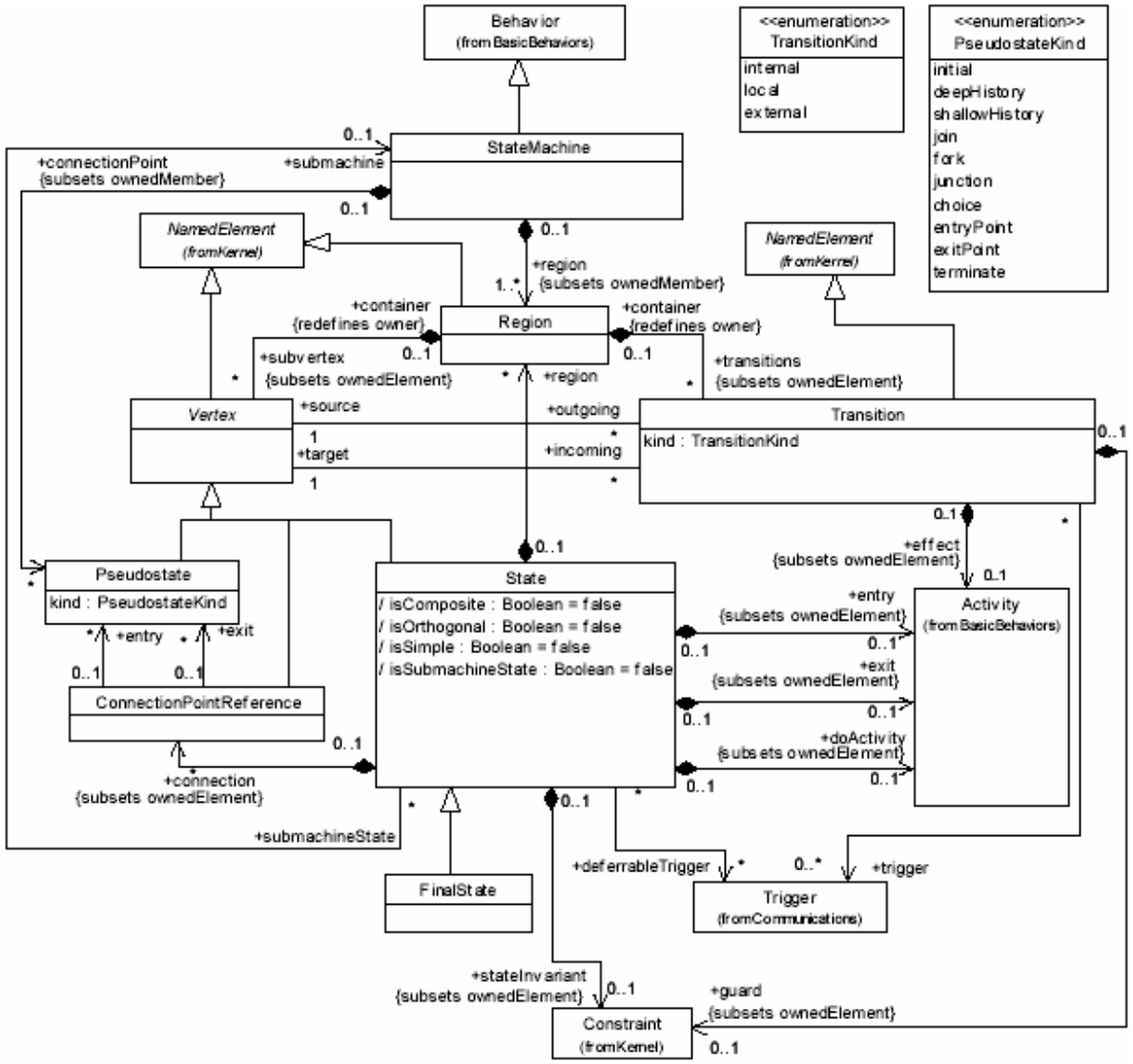


Abbildung 4.8.: Klassendiagramm eines Zustandsautomaten aus dem UML2.0 Metamodell [Uni04]

IFSConstraint-Klasse

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

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

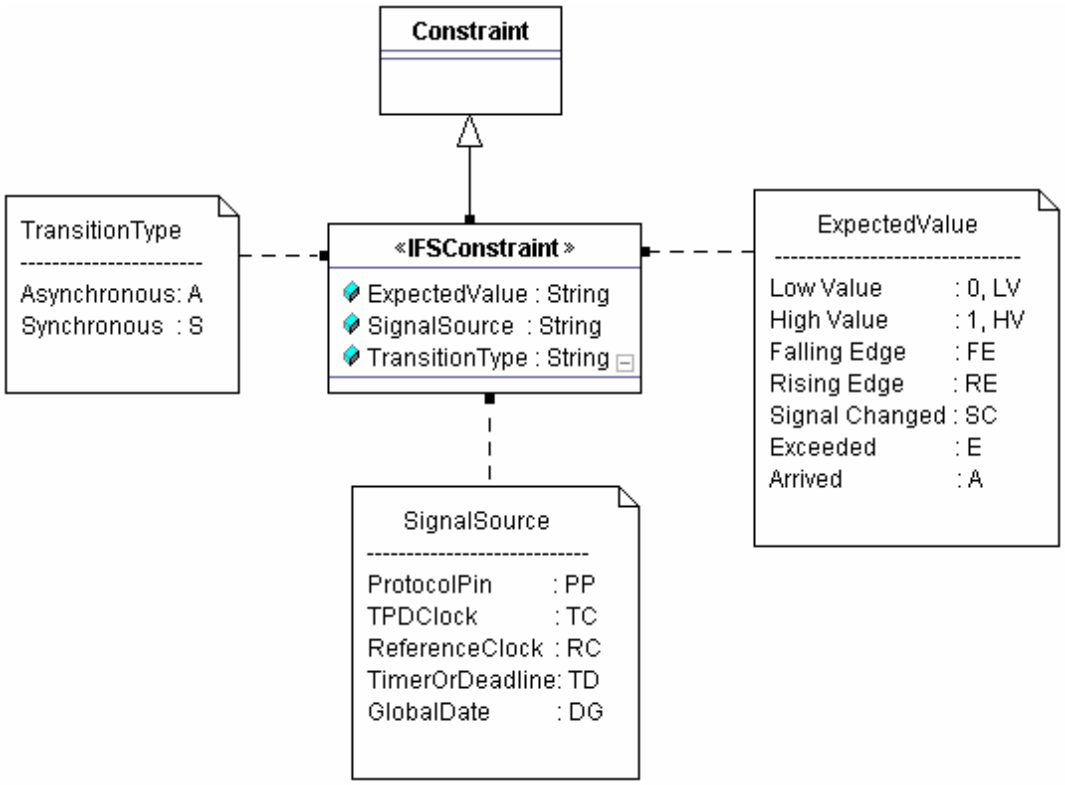


Abbildung 4.9.: Die Klasse "IFSConstraint"

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

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

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

IFSActivity-Klasse

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

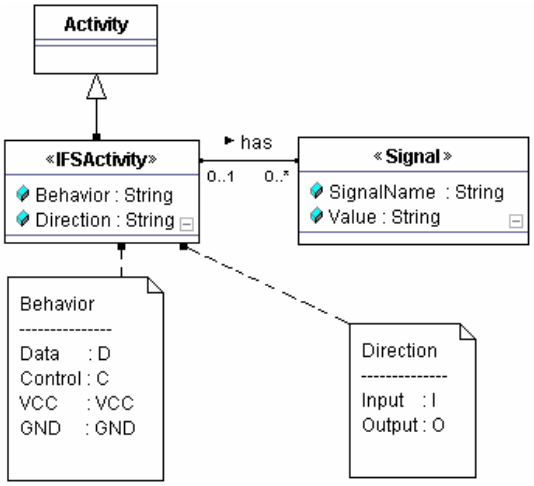


Abbildung 4.10.: Die Klasse “IFSActivity”

State

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

Trigger

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

Signal

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

ReferenceSignal

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

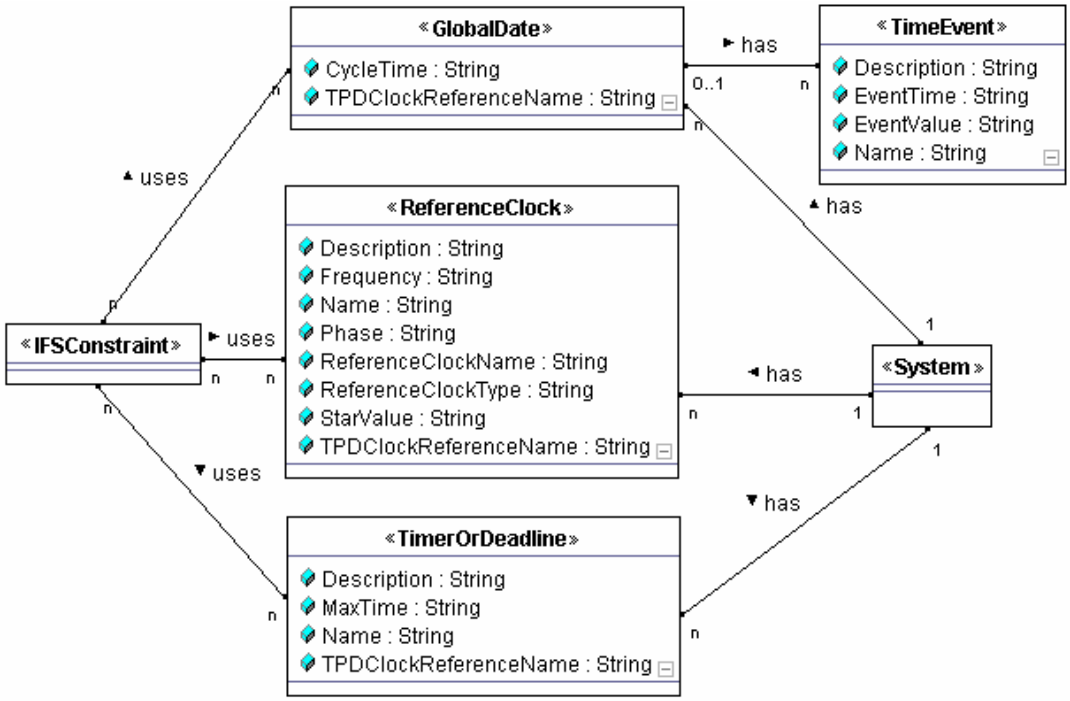


Abbildung 4.11.: Die Klasse "ReferenceSignal"

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

- **CycleTime:**
  Dieses Attribut ermöglicht es, die Periodendauer der zyklischen Zeitbasis zu definieren.

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

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

- **Name und Description:**
  Mit dem Attribut Name werden die einzelnen Signale identifiziert, denen jeweils eine Menge von Ereignissen zugeordnet wird. Ein Ereignis ist ein Tupel aus *EventTime* und *EventValue*. Signale können mit einer *Description* versehen werden.

- **EventTime**
  Mit diesem Attribut wird der Zeitpunkt des Ereignisses innerhalb der *CycleTime* festgelegt.

- **EventValue**
  Das Attribut *EventValue* legt das entsprechende Ereignis (*Low-* oder *High-Pegel*) fest, das zum Zeitpunkt *EventTime* am Signal des entsprechenden *GlobalDates* anliegen soll. Die möglichen Werte sind: *Logic One*, *Logic Zero*, *High Peak*, *Low Peak* und *Signal Change*.

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

- **Frequency:**
  Mit diesem Attribut wird die Frequenz des *Referenzsignals* bestimmt, z.B. Frequency = 1.0E2 [Hz].

- **StartValue:**
  Die Zuweisung eines Anfangswertes erfolgt mit Hilfe des Attributs *StartValue*: Mögliche Werte sind *Logic One* und *Logic Zero*.

- **ReferenceClockName:**
  Der Designer kann an dieser Stelle die Clock referenzieren, die bereits in der zugehörigen *TPD* deklariert ist und aus der die Taktrate abgeleitet wird. Ein Clock-Fehler, der möglicherweise aus Rundungsfehlern resultiert, kann exakt berechnet werden.

- **ReferenceClockType:**
  Der Designer kann mit Hilfe dieses Attributs festlegen, ob es sich um eine Clock der *TPD* oder einer anderen ReferenzClock. handelt. Zur Auswahl sind daher entsprechend Real Clock *(TPD)* und ReferenceClock gegeben.

- **Phase:**
  Diese Angabe gibt die Phasenverschiebung zu einer weiteren ReferenzClock an: *0°, 45° (Pi / 4), 90° (Pi / 2), 135° (3 Pi / 4)* und *180° (Pi)*. Z.B. Phase = Pi/4.

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

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

Target-Platform-Description

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

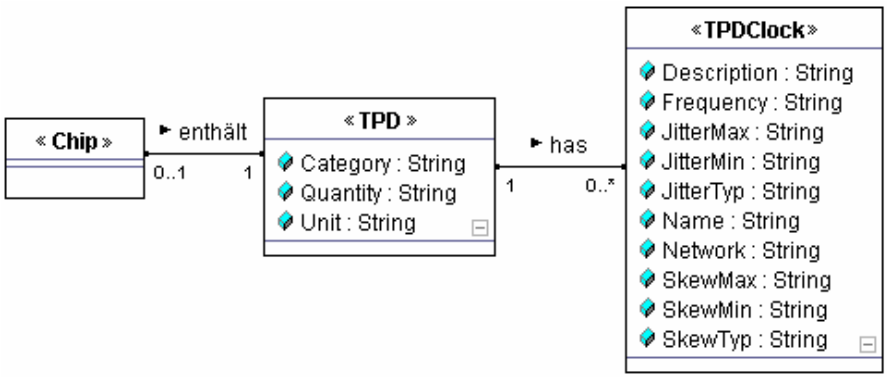


Abbildung 4.12.: TPD Klassendiagramm

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

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

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

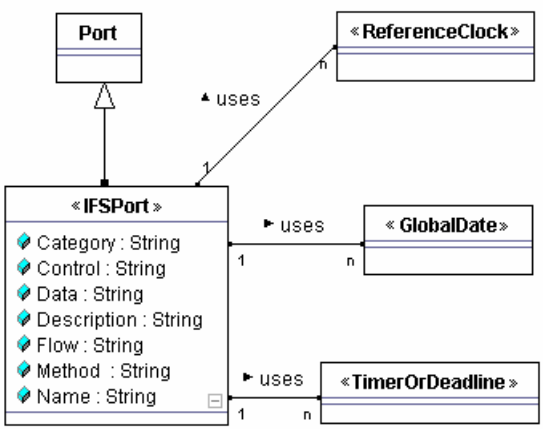


Abbildung 4.13.: Die Klasse "IFSPort"

Protokollattribute und Identifikationsattribute

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

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

5. Konzept und Implementierung der Modelltransformation

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

5.1. Wahl der Transformationsart und des UML Case-Tools

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

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

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

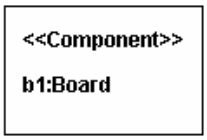


Abbildung 5.1.: Der Stereotype Board

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

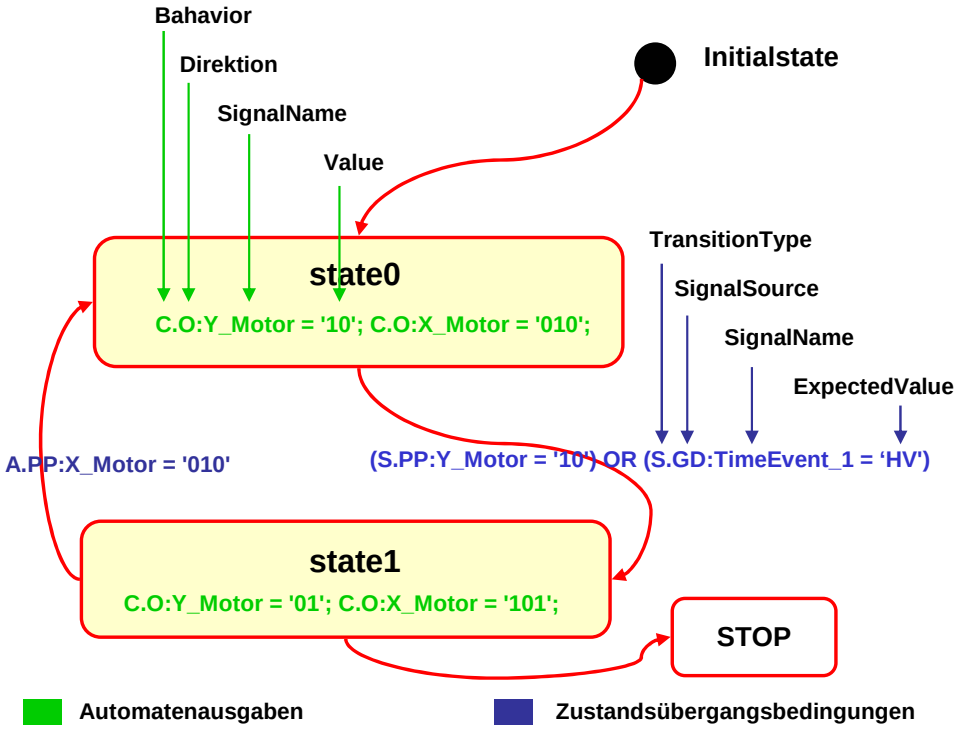


Abbildung 5.2.: Zustandsautomatenmodell

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

5.2. Transformationskonzept

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

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

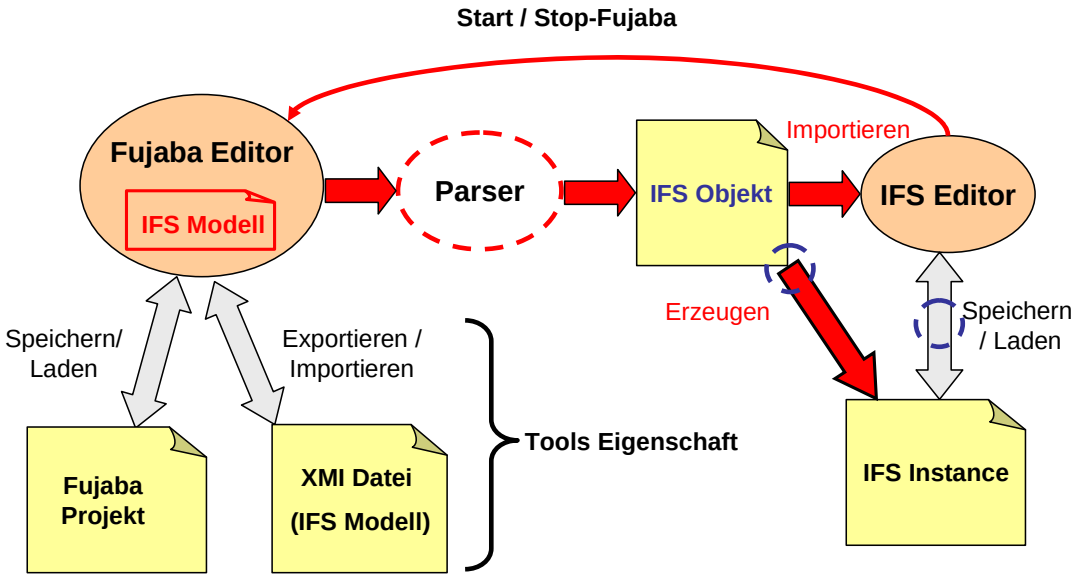


Abbildung 5.3.: Transformationskonzept

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

5.3. Implementierung der Modelltransformation

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

- Werkzeug-Kopplung
- Parseralgorithmus

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

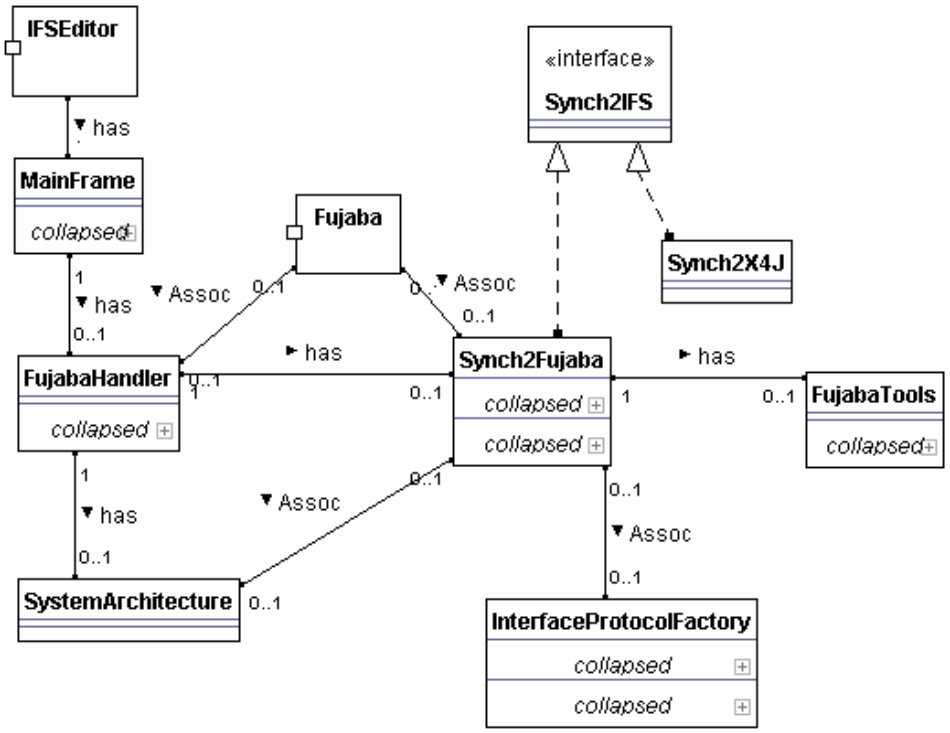


Abbildung 5.4.: Klassendiagramm der implementierten Modelltransformation

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

5.3.1. Werkzeug-Kopplung

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

Abbildung 5.5.: IFS-Editor: Start Fujaba

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

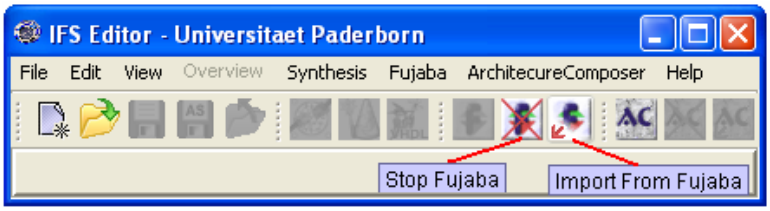


Abbildung 5.6.: IFS-Editor: Import From und Stop Fujaba

MainFrame

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

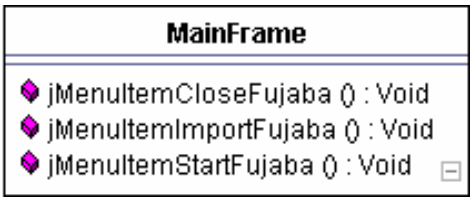


Abbildung 5.7.: MainFrame

FujabaHandler

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

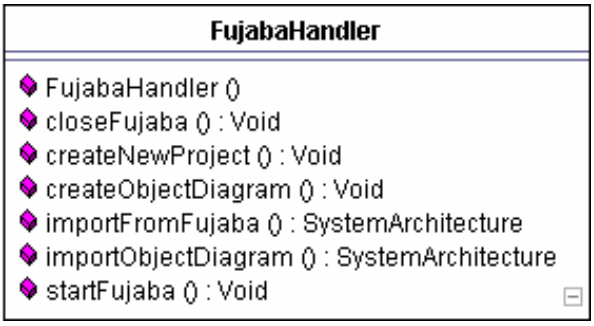


Abbildung 5.8.: FujabaHandler

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

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

5.3.2. Parseralgorithmus

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

Synch2Fujaba

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

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

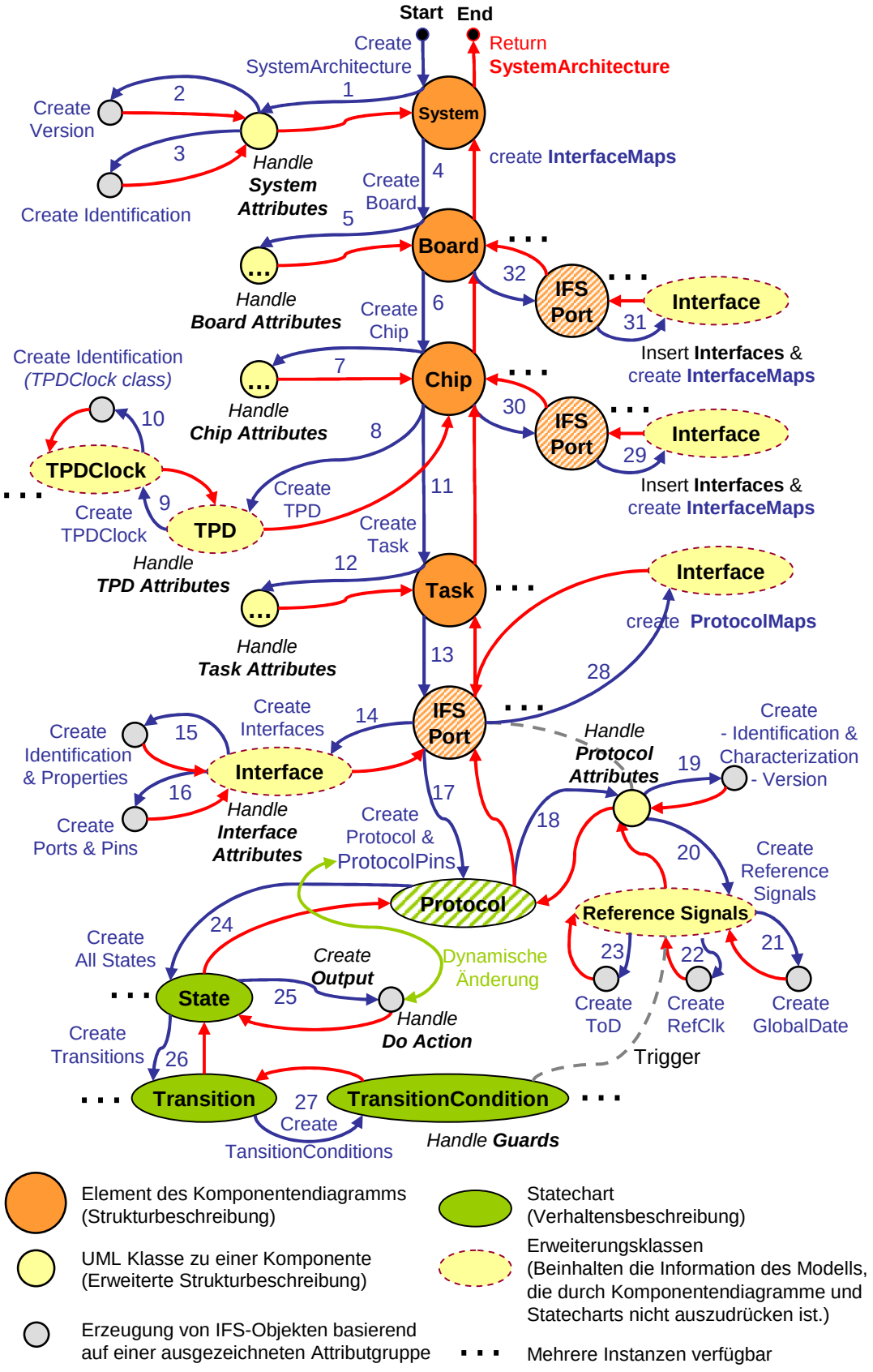


Abbildung 5.9.: Rekursiver Abstieg des Parseralgorithmus

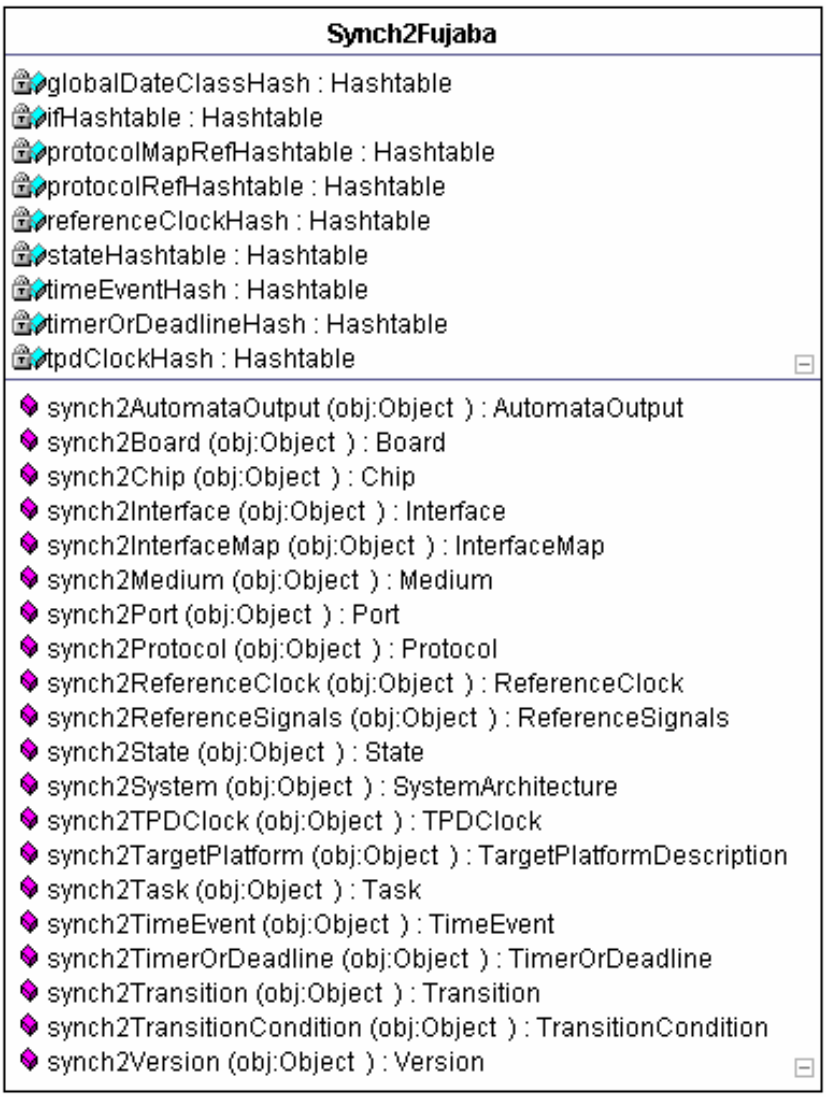


Abbildung 5.10.: Die Klasse “Synch2Fujaba”

System *(Synch2System())*

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

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

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

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

Boards (*synch2Board()*)

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

Chip (*synch2Chip()*)

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

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

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

Task (*synch2Task()*)
Synch2Task() ist eine komplexe Methode, die zusätzlich bei der Erzeugung des IFS-Objektes *"Task"* neben den zugehörigen Attributen und der Version, die Protokollaspekte des modellierten Zustandautomaten berücksichtigt. Für die Erzeugung des Protokolls werden zuerst alle Ports der Task (UML-Objekte) gesucht und einzeln wie folgt abgearbeitet. Die Referenzen eines Ports auf seine *ProvidedInterfaces* bzw. *RequiredInterfaces* (UML-Objekt) werden ermittelt und dann dem Methodenaufruf synch2Interface() übergeben. Dabei ist zu beachten, dass im IFS-Modell genau ein Interface pro Port erlaubt ist (siehe Kapitel 4.2.1).

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

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

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

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

- **Protocol *(synch2Protocol())***
  Wie bei allen *synch()* Methodeaufrufen wird auch hier ein IFS-Objekt (*"Protocol"*) angelegt. Im Gegensatz zu den anderen *synch()* Methoden wird das *Protocol* durch die Methode *generateProtocolFromInterface()* der InterfaceProtocolFactory-Klasse erzeugt. Bei dem Aufruf der *generateProtocolFromInterface()* Methode wird die Referenz des bereits erzeugen Interfaces übergeben. Aus den vorhandenen Port-Interfaces wird das IFS-Objekt *"ProtokollPins"* generiert (siehe Abbildung 5.11).

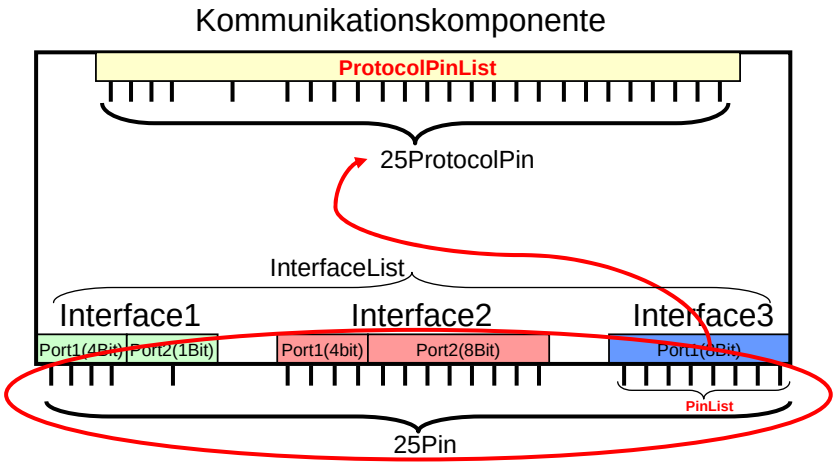


Abbildung 5.11.: Generierung der ProtocolPins

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

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

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

  AutomataOutput *(synch2AutomataOutput())*

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

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

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

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

InterfaceProtocolFactory

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

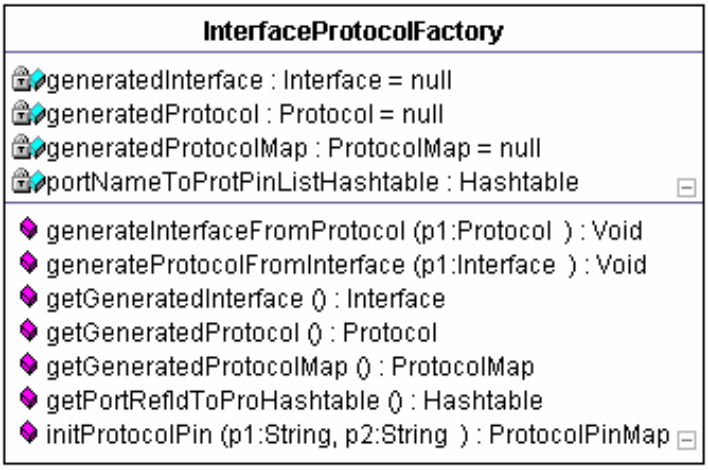


Abbildung 5.12.: Die Klasse “InterfaceProtocolFactory”

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

    generatedProtocol = new Protocol(newDefaultID);
    generatedProtocol.setName(inter.getName() + "_GenPr");
    generatedProtocol.setDescription("Generated␣Protocol␣from␣Interface␣"
            + inter.getName() + "␣(ID:␣" + inter.getID() + ")");

    generatedProtocolMap = new ProtocolMap();
    generatedProtocolMap.setID(newDefaultID);
    generatedProtocolMap.setName("I" + inter.getID() + "_P_GenPrMap");
    generatedProtocolMap.setDescription("Generated␣Map␣from␣Interface␣"
            + inter.getName() + "␣(ID:␣" + inter.getID() + ")");
    generatedProtocolMap.setInterfaceID(inter.getID());
    generatedProtocolMap.setProtocolID(generatedProtocol.getID());

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

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

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

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

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

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

FujabaTools

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

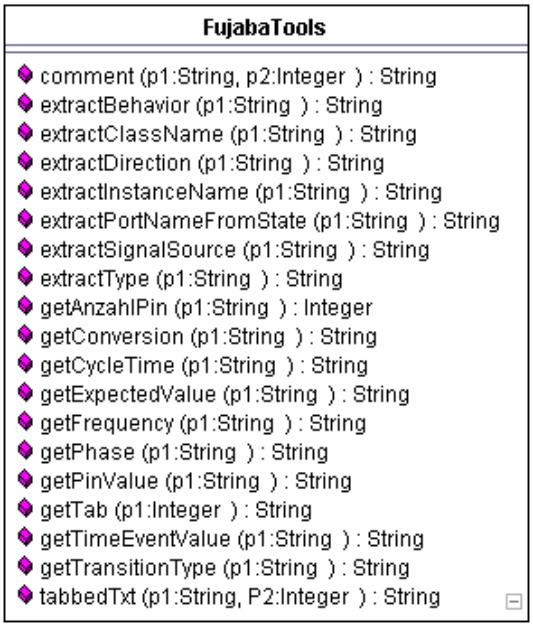


Abbildung 5.13.: Die Klasse “FujabaTools”

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

extractPortNameFromState()

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

extractDirection()

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

extractBehavior()

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

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

getPinValue()

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

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

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

extractClassName(), extractInstanceName()

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

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

getTimeEventValue()

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

```
/*
*  return "High Peak" für den String "HP" bzw. "H" ...
*/
public static String getTimeEventValue(String string)
{
    if (string.equals("1"))
        return "Logic One";
    else if (string.equals("0"))
        return "Logic Zero";
    else if (string.startsWith("H") || string.startsWith("h"))
        return "High Peak";
    else if (string.startsWith("L") || string.startsWith("l"))
        return "Low Peak";
    else if (string.startsWith("S") || string.startsWith("s"))
        return "Signal Changed";
    else
        return null;
}// getTimeEventValue
```

extractType()

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

Comment(), tabbedTxt() und getTab()

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

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

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

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

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

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

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

6.2. Ausblick

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

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

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

A. Anhang

A.1. Beschreibung eines Beispielmodells

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

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

A.1.1. Die grafische Oberfläche

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

A.1.2. Komponente und Objektdiagramm

Komponente

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

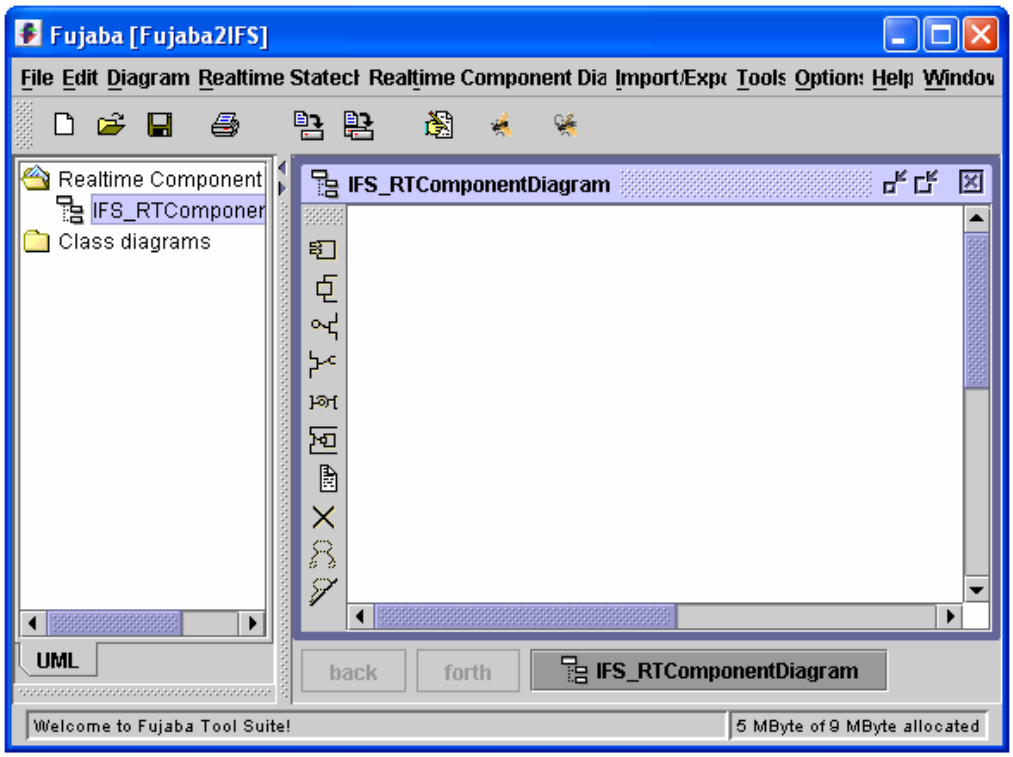


Abbildung A.2.: Fujaba Editor

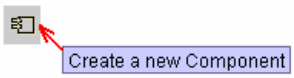


Abbildung A.3.: Knopf “Create a new Component”

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

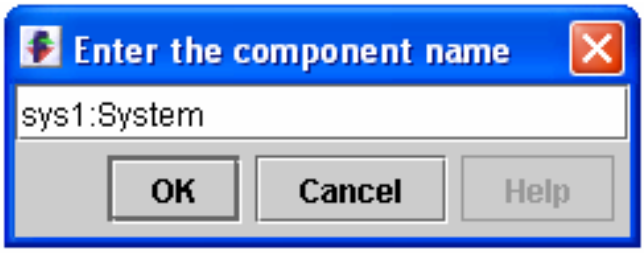


Abbildung A.4.: Maske zur Eingabe des Komponentennamens

Objektdiagramm

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

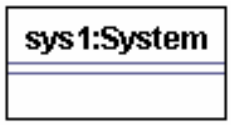


Abbildung A.5.: Das Objektdiagramm "sys1System"

Eingabe von Attributen

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

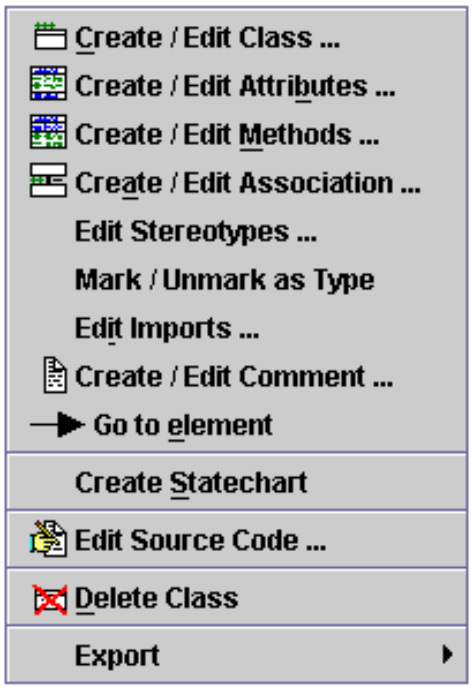


Abbildung A.6.: Kontextmenü für ein Klassen- bzw. Objektdiagramm

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

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

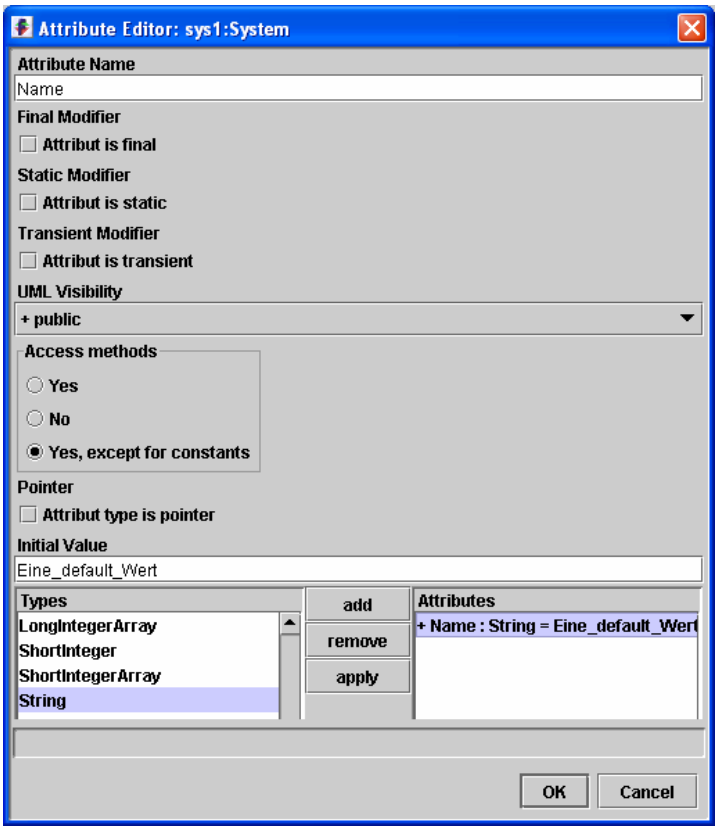


Abbildung A.7.: Eingabefenster für die Attribute

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

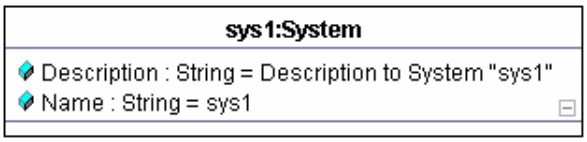


Abbildung A.8.: Attribute der Klasse "System"

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

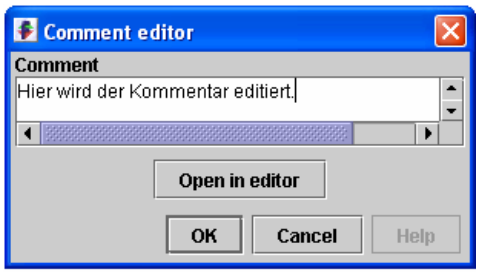


Abbildung A.9.: Kommentarfenster

Subkomponenten

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

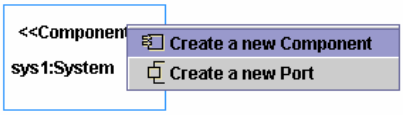


Abbildung A.10.: Erzeugung von Subkomponenten

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

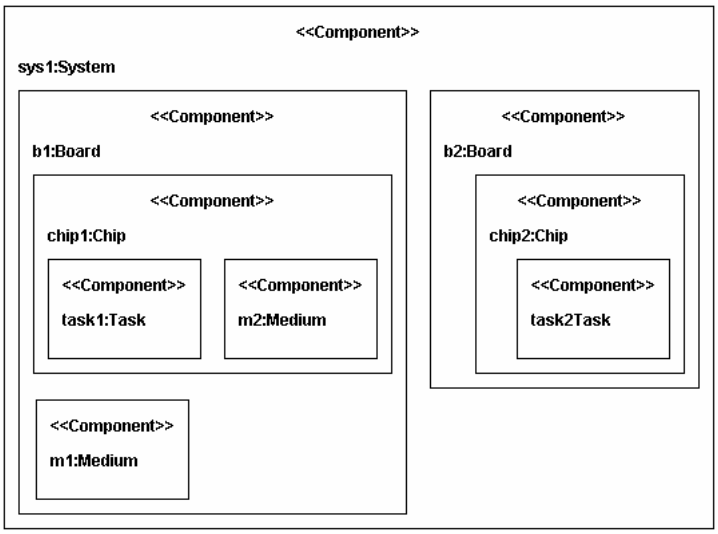


Abbildung A.11.: Komponenten der Systemarchitektur

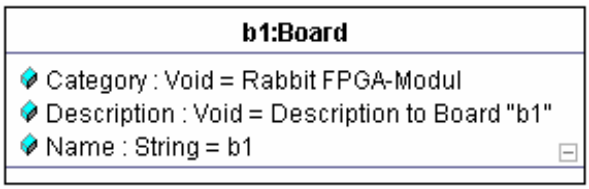


Abbildung A.12.: Instanz der Komponente Board

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

Board-Kategorie

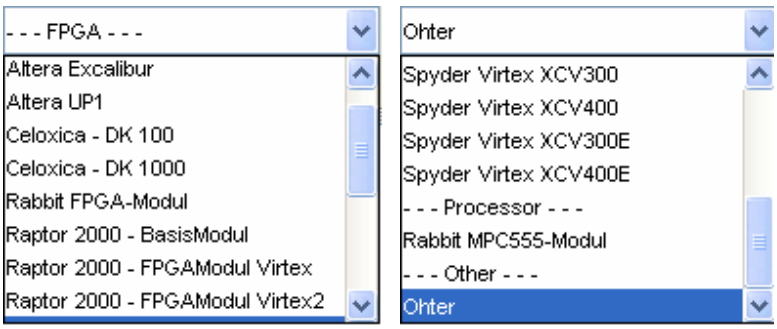


Abbildung A.13.: Board-Kategorie

Board-Kategorie

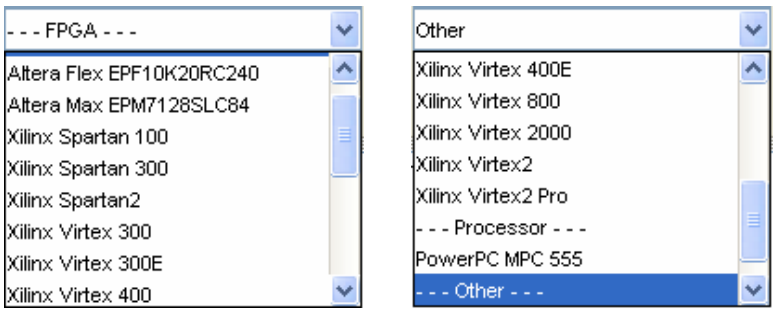


Abbildung A.14.: Chip-Kategorie

Task-Kategorie

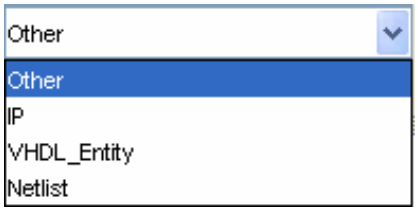


Abbildung A.15.: Task-Kategorie

Medium-Kategorie

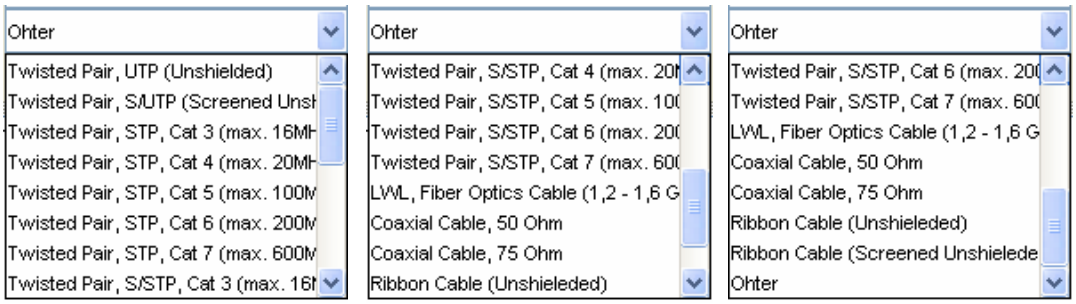


Abbildung A.16.: Medium-Kategorie

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

Interface-Klasse

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

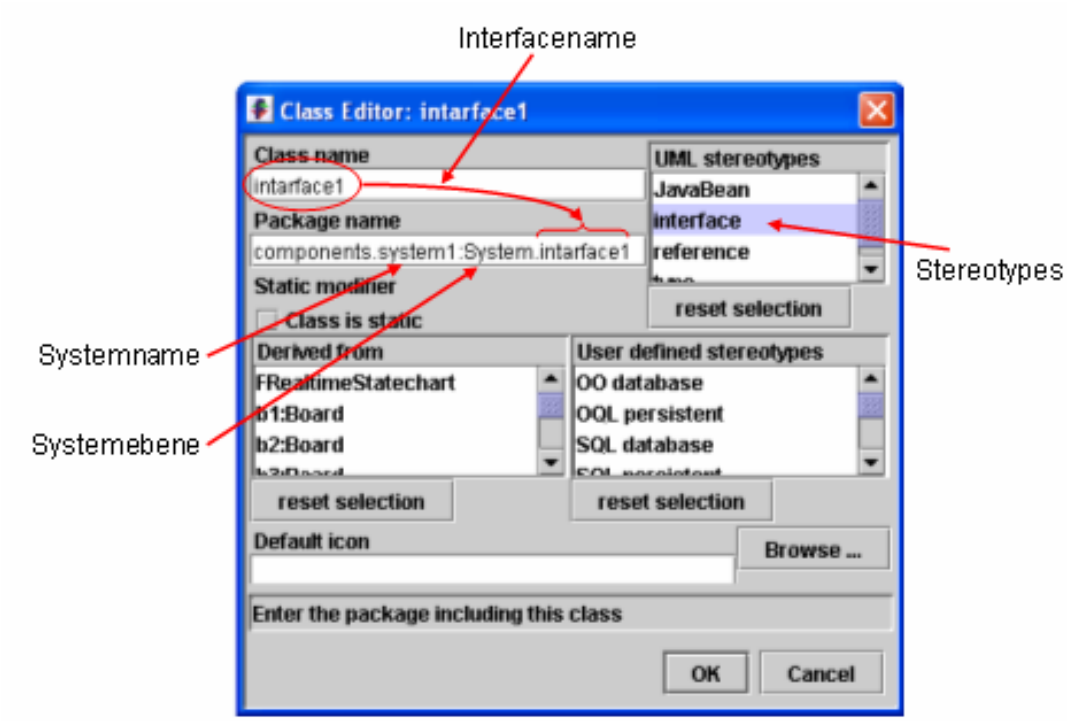


Abbildung A.17.: Interface Klassendeklaration

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

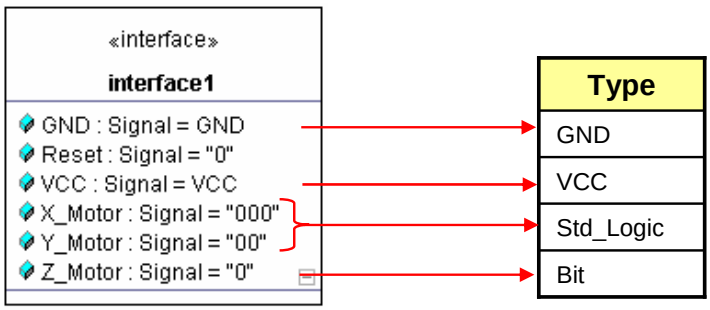


Abbildung A.18.: Signaldeklaration in der Interface-Klasse

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

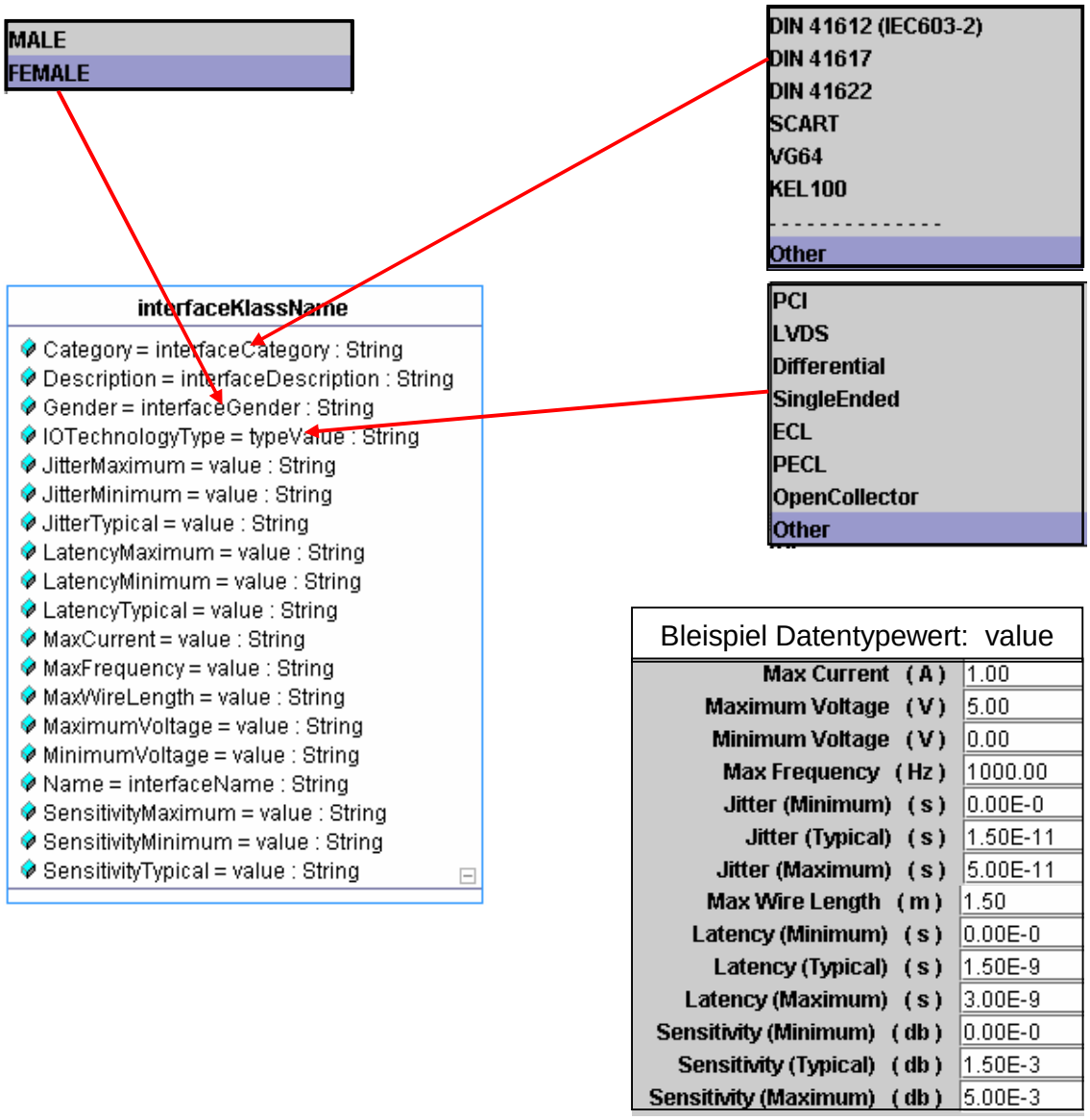


Abbildung A.19.: Mögliche Attributwerte der Interface-Klasse

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

Schnittstelle

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

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

Abbildung A.20.: Erzeugung eines Ports

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

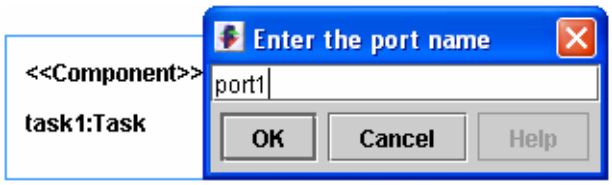


Abbildung A.21.: Auswahl des Portnamens

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

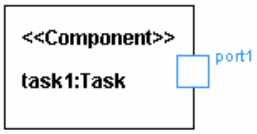


Abbildung A.22.: Beispiel-Port

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

Abbildung A.23.: Portinstanz

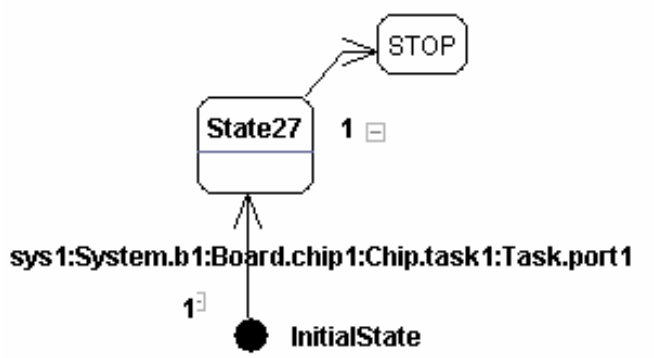


Abbildung A.24.: Zustandsautomaten für den Port “port1”

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

Abbildung A.25.: Attributeswerte einer Portinstanz

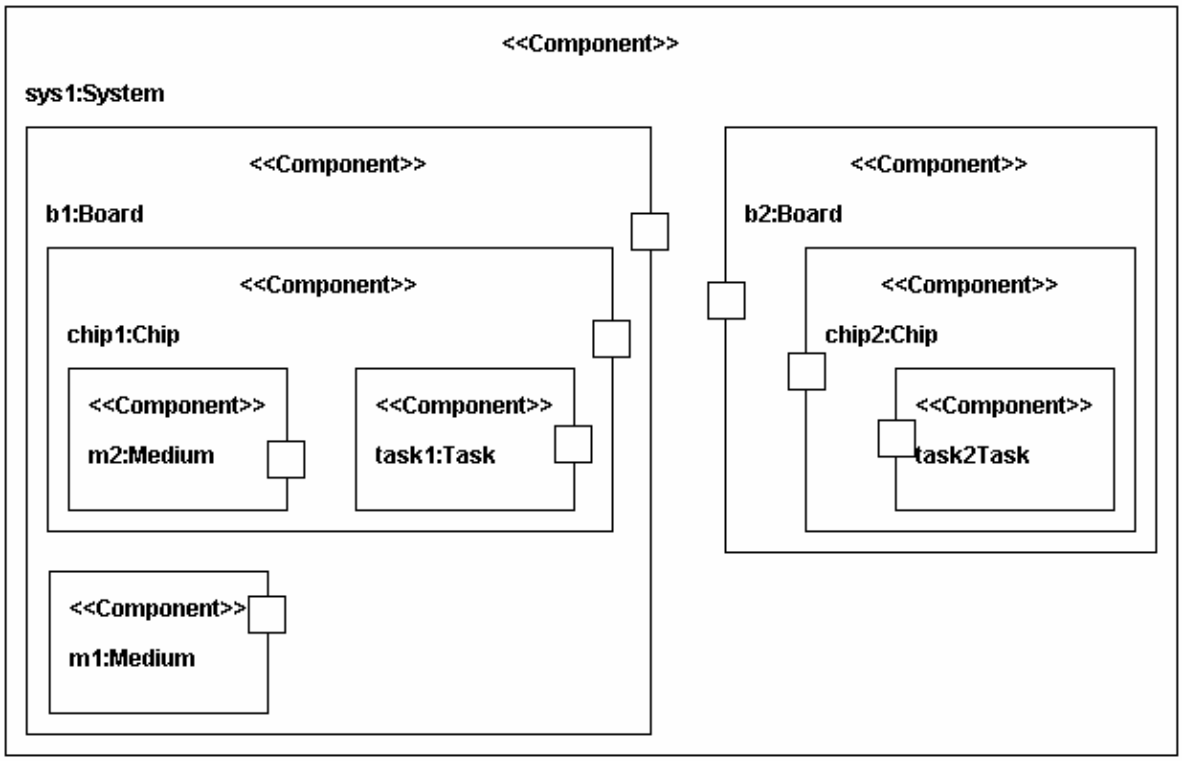


Abbildung A.26.: Komponenten und Ports der Systemarchitektur

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

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

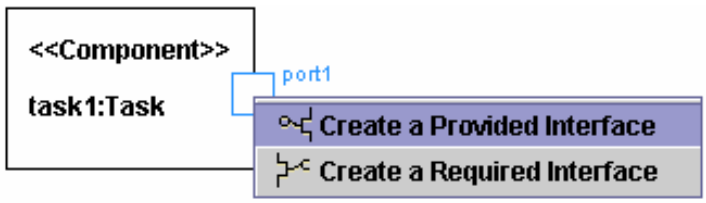


Abbildung A.27.: Erzeugen von Provided- oder Required Interfaces

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

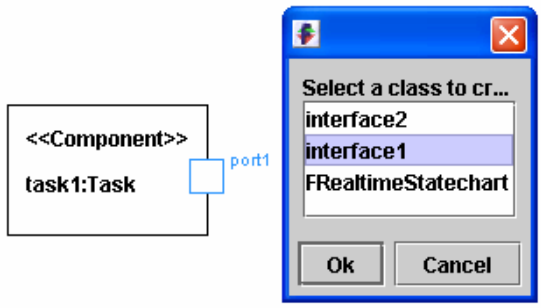


Abbildung A.28.: Auswahl des implementierten Interfaces

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

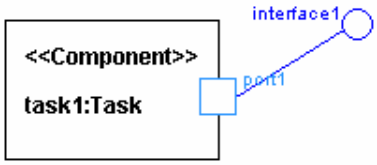


Abbildung A.29.: Referenzinterface

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

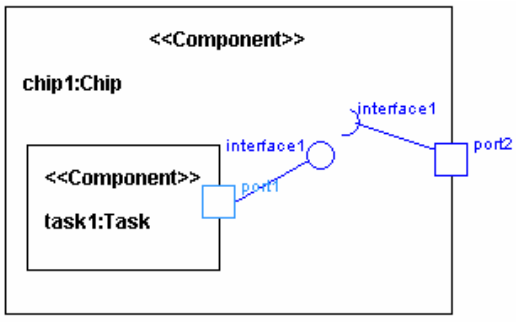


Abbildung A.30.: Provided und Required Interface

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

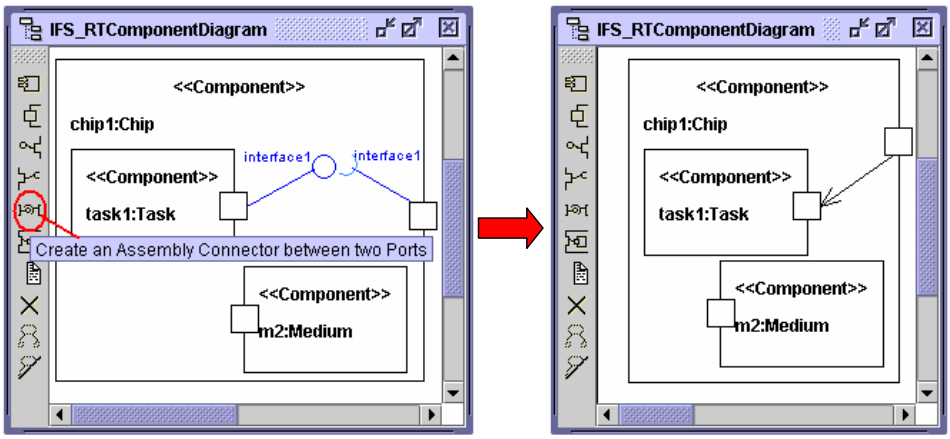


Abbildung A.31.: Connector

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

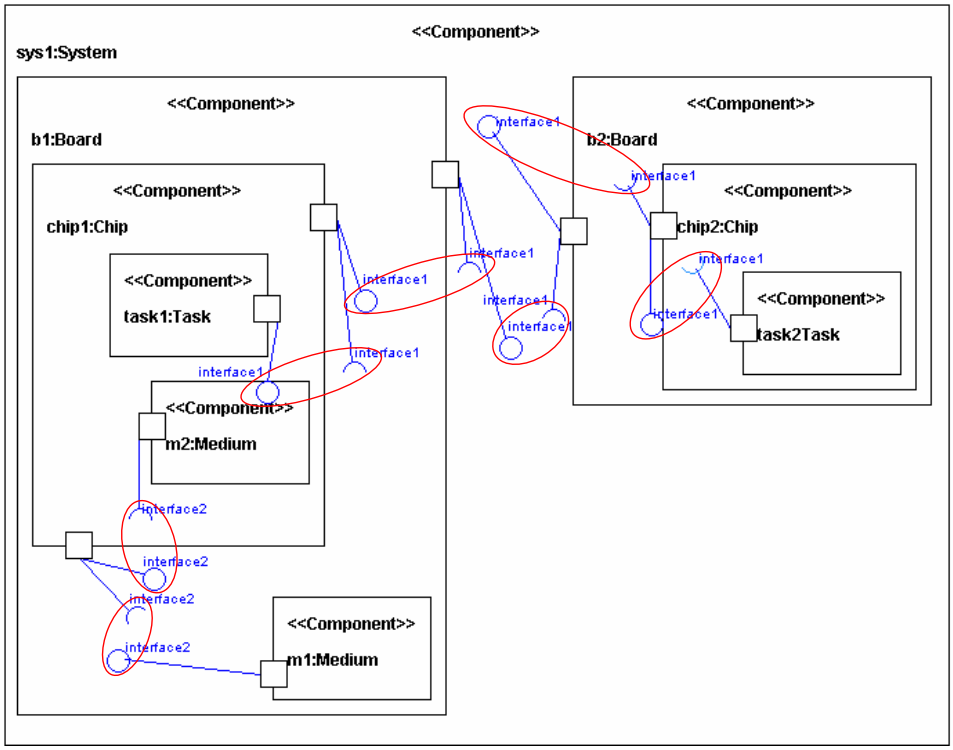


Abbildung A.32.: Erstellung der Connectoren

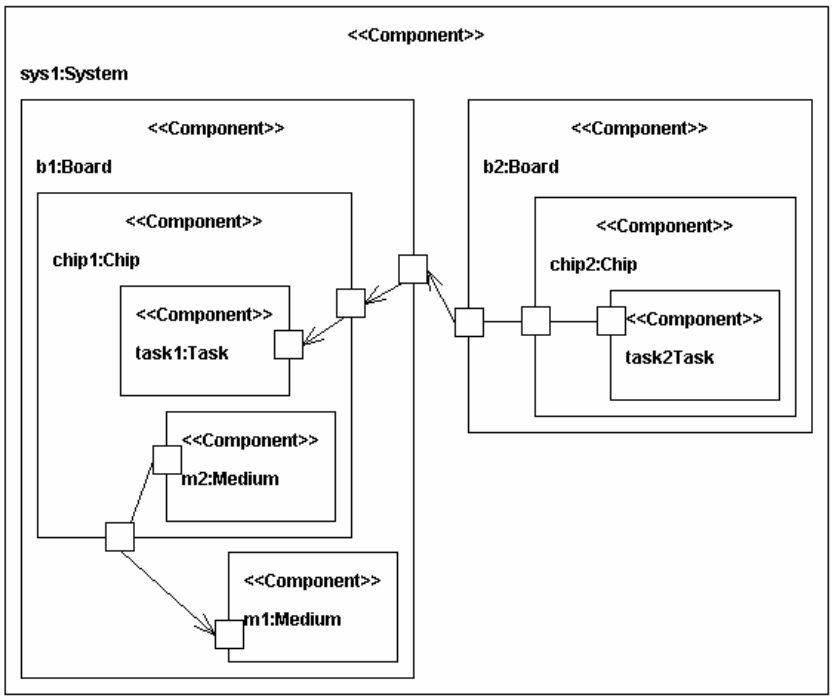


Abbildung A.33.: Gesamte Kommunikationsverbindungen

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

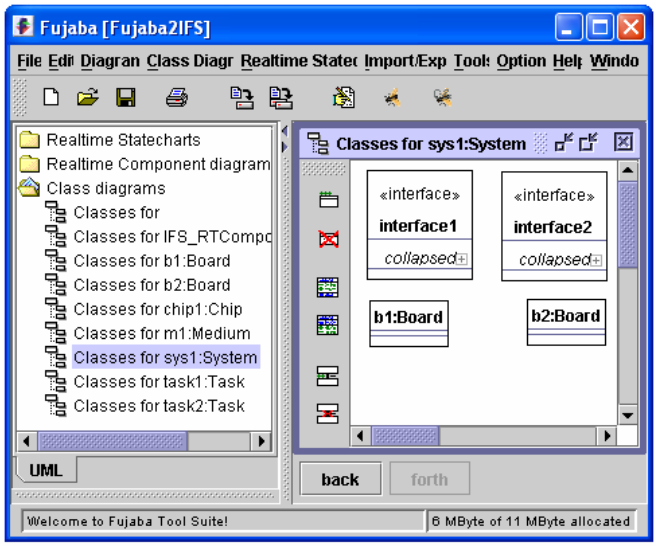


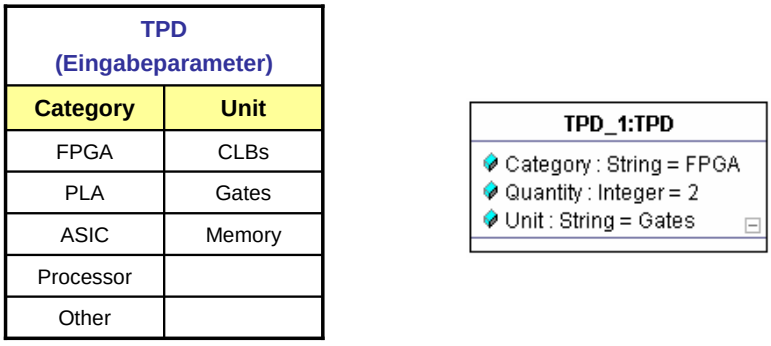
Abbildung A.34.: Systemebene-Klassendiagramme

A.1.3. Verhaltensbeschreibung

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

TPD

Die Abbildung A.35 stellt auf der linken Seite eine Tabelle dar. Die Tabelle beinhaltet die möglichen Werte für die Attribute Category und Unit des TPDs. Auf der rechten Seite ist die modellierte TPD-Instanz *"TPD_1"* zu sehen. Die *TDP_1* wurde in der Chipebene *"Classes for chip1:Chip"* (siehe Abbildung A.34) erzeugt.

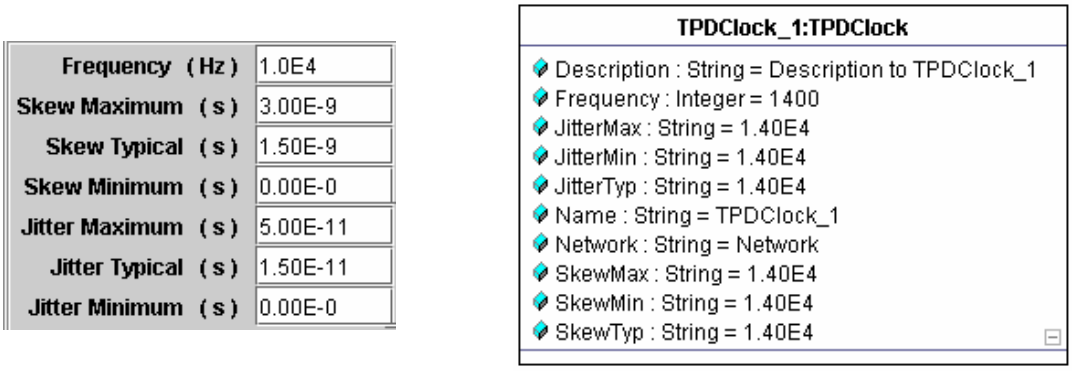


TPD (Eingabeparameter)	
Category	**Unit**
FPGA	CLBs
PLA	Gates
ASIC	Memory
Processor	
Other	

Abbildung A.35.: TPD-Instanz

TPDClock

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



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

Abbildung A.36.: TPDClock-Instanz

ReferenceClock (Eingabeparameter)			
Phase	**Type**	**StartValue**	**Frequency**
0 Pi	Real Clock (TPD)	Logic One	ZB. 1.0E0
¼ Pi	ReferenceClock	Logic Zero	
½ Pi			
¾ Pi			

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

Referenzsignal

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

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

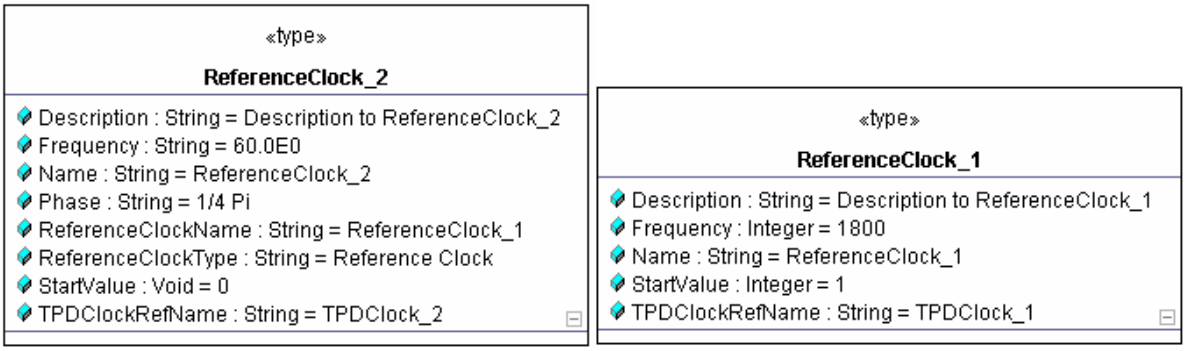


Abbildung A.37.: ReferenceClock-Instanz mit und ohne Phasenverschiebung

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

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

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

Tabelle A.2.: Mögliche Werte des Attributs *"EventValue"*

Abbildung A.38.: TimeEvent-Instanz

- **GlobalDate**
  Eine Instanz der GlobalDate-Klasse *"GlobalDate_ 1"* wird auf der Systemebene modelliert (siehe Abbildung A.39). Das *GlobalDate_ 1* hat zwei Attribute vom Typ TimeEvent, das sind *TimeEvent1* und *TimeEvent2*. Die beiden Attribute haben als Instanz den Namen der vorher modellierten TimeEvent-Instanz (siehe Abbildung A.38). Die Namen werden für den Parseralgorithmus als Referenz verwendet. Mit diesen Namen werden die entsprechenden TimeEvents in der Systemebene gesucht.

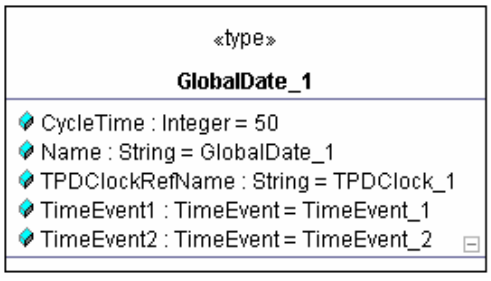


Abbildung A.39.: GlobalDate-Instanz

- **TimeOrDeadline**

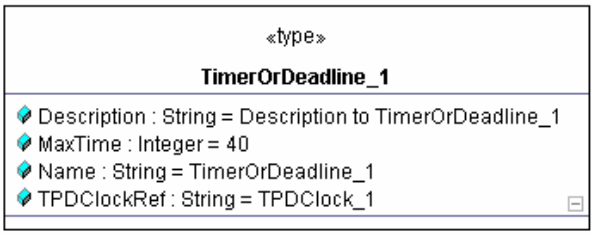


Abbildung A.40.: TimeOrDeadline-Instanz

A.1.4. Zustandsautomaten

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

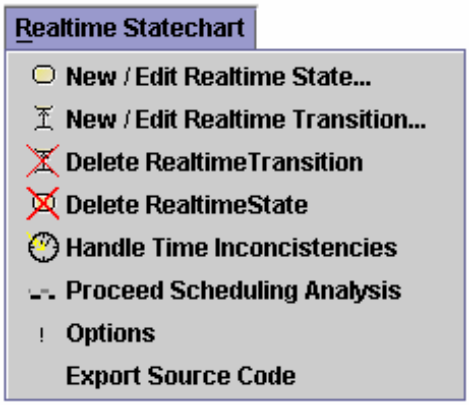


Abbildung A.41.: Auswahlmenü für das Editieren von Zustandsautomaten

State

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

TransitionType	Kennung
Asynchronous	A
Synchronous	S

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

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

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

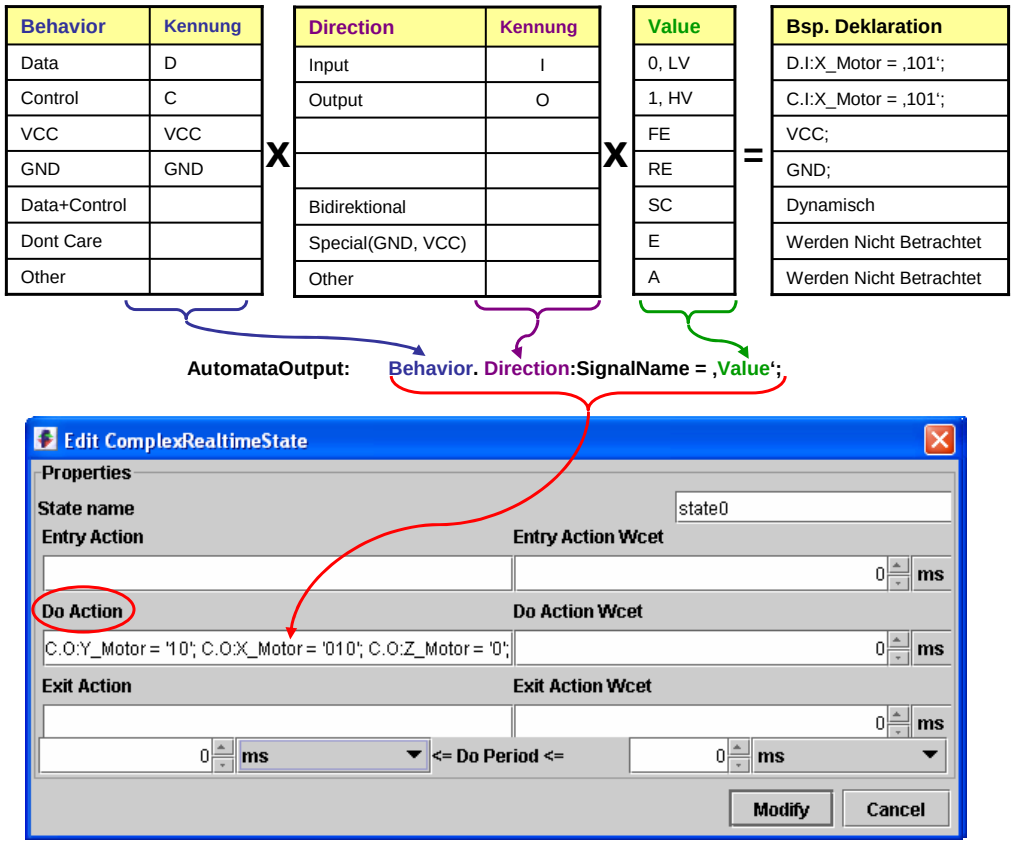


Abbildung A.42.: Eingabefenster für ein AutomataOutput: *“Do Action”*

Transition

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

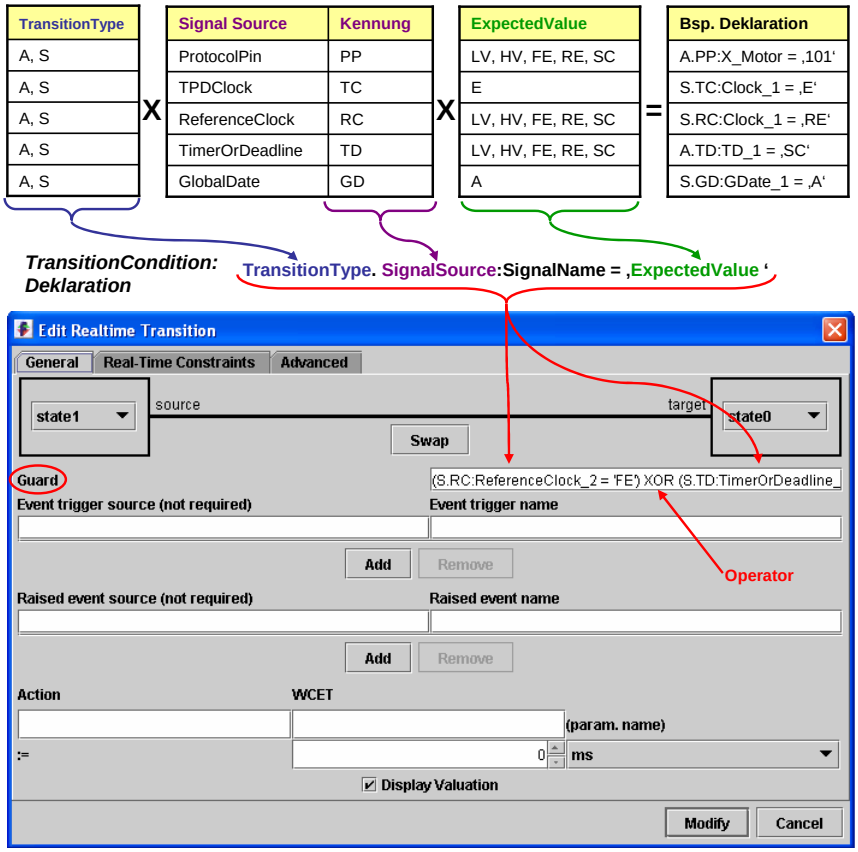


Abbildung A.43.: Eingabefenster für die Zustandsübergangsbedingungen "Guard"

Operator
AND
OR
XOR
NAND
NOR
XOR

Abbildung A.44.: Mögliche Werte für die Operatoren

Beispiel eines Zustandsautomaten

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

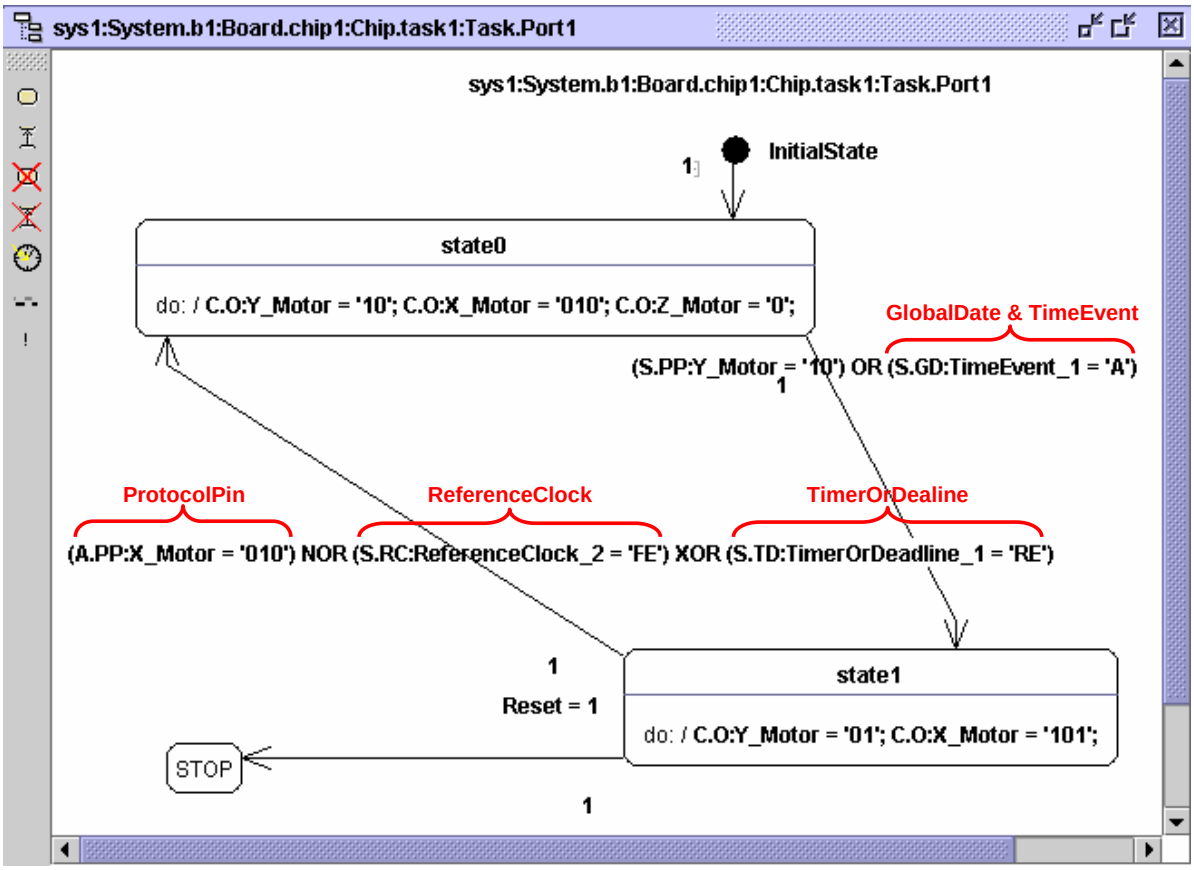


Abbildung A.45.: Modellierter Zustandsautomat für den Port "port1"

A.2. Beispiele zur XML Transformation

A.2.1. SAX-Parser-Beispiel

SaxPaserDemo [Bre01]

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

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

    public void performDemo(String uri){
        System.out.println("XML-Datei wird geparst: " + uri + "\n");
        try{
        // SAX Parser instanziieren
        XMLReader parser = new SAXParser();
        // Das Dokument parsen
        MyContentHandler myContentHandler = new myContentHandler();
        // Setzen des implementierten Interfaces(ContentHandler)
        parser.setContentHandler(myContentHandler);
        parser.parse(uri);
        }catch (IOException e){
            System.out.println("Fehler beim Lesen des URI: "
                                + e.getMessage());
        }catch (SAXException e){
            System.out.println("Fehler beim Parsen: " + e.getMessage());
        }
    }

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

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

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

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

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

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

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

A.2.2. DOM-Parser-Beispiel

DOMPaserDemo [Bre01]

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

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

```
    public void performDemo(String uri){
        System.out.println("XML-Datei wird geparst: " + uri + "\n\n");
        // DOM-Parser instantieren
        DOMParser parser = new DOMParser();
        try{
        /**
          Das Dokument parsen. Hier gibt es unterschiedlichen
          DOM-Implementierungen der Methode parse() auf org.w3c.dom.
          Die Methode liefert entweder ein Document-Objekt zurück
          oder stellt eine Methode getDocument() zur Verfügung.
         */
        parser.parse(uri);
        Document docu = parser.getDocument();
        // Den eingelesenen DOM-Baum ausgeben
        printNode(docu);
        }catch (Exception e){
            System.out.println("Fehler beim Lesen des URI: " + e.getMessage());
        }
    }
    // Die Interface Methode können überschrieben werden um
    // die DOM-Struktur zur verändern oder das Dokument
    // als DOM-Node-Objekt verwenden.
        public void printNode(Node node){
            // Den Typ des Knotens bestimmen
            // den knoten bearbeiten und ausgeben
            // rekursiv auf den Kindknoten arbeiten
        }

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

A.3. Java Quellcode

A.3.1. Synch2System()

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

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

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

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

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

Enumeration enum = sysConnectorHash.keys();
```

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

    Port portFrom = connector.getPortFrom();
    Port portTo = connector.getPortTo();
    if (portOfSubCompHash.containsKey(portFrom.getID())
        && portOfSubCompHash.containsKey(portTo.getID()))
    {
        de.upb.ipl.ifds.components.Interface interA = (de.upb.ipl.ifds.
            components.Interface) ifRefHashtable.get(portFrom.getID());
        de.upb.ipl.ifds.components.Interface interB = (de.upb.ipl.ifds.
            components.Interface) ifRefHashtable.get(portTo.getID());
        // InterfaceMapList
        try
        {
          if (interA.getParent() != null && interB.getParent() != null)
          {
            InterfaceMap interMap = new InterfaceMap(interA, interB);
                system.addInterfaceMap(interMap);
          }
        }
        catch (Exception e)
        {
          if (DEBUG > 0)
            System.out.println(FujabaTools.getTab(3) + "FATAL ERROR: "
            + "Program code wrong in InterfaceMap.java\n(method: " +
            "InterfaceMap(Interface interA, Interface interB)). " +
            "At least one parent was null.");
        }
    }
 } // while
 return system;
}//synch2System
```

A.3.2. synch2Protocol()

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

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

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

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

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

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

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

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

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

```
        }
      }
      return generatedProtocol;
} //synch2Protocol
```

A.3.3. extractType()

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

Literaturverzeichnis

[Alf99] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilerbau, 2. Teil.* R.Oldenbourg Verlag München Wien, Dezember 1999.

[Alf02] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* 2002.

[And04] Andreas Andresen. *Komponentenbasierte Softwareentwicklung mit MDA, UML 2 und XML*, volume 2. Auflage. Hanser fachbuchverlag edition, August 2004.

[Bec03] Beckert, Bernhard. *Formale Spezifikation und Verifikation von Software.* Universität Koblenz-Landau, http://www.uni-koblenz.de/∼beckert/Lehre/Spezifikation-Verifikation/skriptum-schmitt.ps.gz, Wintersemester 2002/2003.

[Ber04] Bernd, Oestereich. *Objektorientierte Softwareentwicklung - Analyse und Design mit der UML 2*, volume 6. überarbeitete Auflage. R.Oldenbourg Verlag München Wien www.oose.de/uml, Februar 2004.

[Boo] Booch, Grady and Rumbaugh, Jim and Jacobson, Ivar. *Das UML Benutzerhandbuch mit Bleispielen zunehmender Komplexität.*

[Bor] Borland. *Object International. Together J, the Together J case tool. Online at.* http://www.togethersoft.com.

[Bre01] Brett, Mclaughlin. *Deutsche Übersetzung von Matthias Kalle Dalheimer, Java und Xml*, volume 1. Auflage. by o'reilly verlag edition, 2001.

[Fic03] Fick, Oliver. Visualisierung von Parametern komplexer Schnittstellen für eingebettete Systeme auf Basis von XML. Master's thesis, Universität Paderborn, Warburger Str. 100, 33098 Paderborn, June 2003.

[Hol03] Holger Giese and Matthias Tichy and Sven Burmester and Wilhelm Schäfer and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[Ihm01] Ihmor, Stefan. Entwurf von Echtzeitschnittstellen am Beispiel interagierender Roboter. Master's thesis, Universität Paderborn, Warburger Str. 100, 33098 Paderborn, November 2001.

[Ihm02] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. A Consistent Design Methodology for Configurable HW/SW-Interfaces in Embedded Systems. In *Proc. of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems*, Montreal, Canada, Aug. 2002.

[Jec04a] Jeckle, Mario. *Unified Modeling Language (UML) Tools.* http://www.jeckle.de/umltools.htm, Juni, 2004.

[Jec04b] Jeckle, Mario and Rupp, Chris and Hahn, Jürgen and Zengler, Barbara and Queins, Stefan. *UML2 glasklar.* www.uml-glasklar.de, carl hanser verlag münchen wien edition, 2004.

[Pat03] Patrick Grässle, Henriette Baumann, Philippe Baumann. *UML projektorientiert - Ausblick auf den neuen Standard 2.0 - Galileo Computing*, volume 1. Auflage. Mai 2003.

[Pat04] Patrick Grässle, Henriette Baumann, Philippe Baumann. *UML 2.0 projektorientiert - Geschäftsprozessmodellierung, IT-System-Spezifikation und Systemintegration mit der UML, mit Farbposter*, volume 3. Auflage. Juli 2004.

[Rat] Rational. Rose. The Rational Rose case tool. http://www.rational.com.

[Rat99] Rational Software Corporation. *UML documentation version 1.3. Online at.* http://www.rational.com, 1999.

[Rei98] Reinhard Wilhelm, Dieter Maurer. *Übersetzerbau*, volume 2. überarbeitete und erweitete Auflage. Springer, Berlin, Oktober 1998.

[Sve04] Sven Burmester and Holger Giese and Jörg Niere and Matthias Tichy and Jörg P. Wadsack and Robert Wagner and Lothar Wendehals and Albert Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, August 2004.

[Tho04] Thomas Erler. *Das Einsteigerseminar UML 2.* Vmi buch edition, November 2004.

[Uni] Universität Paderborn. *Fujaba - From Uml to Java And Back Again.* http://wwwcs.upb.de/cs/fujaba/.

[Uni04] Unified Modeling Language. *Superstructure version 2.0 Online at.* http://www.omg.org/docs/ptc/03-08-02.pdf, Published on April 30, 2004.

[Vir01] Virtual Component Transfer Development Working Group. *Virtual Component Transfer Specification Version 2.1.* VSI Alliance^{TM}, vct 1 2.1 edition, January 2001.

[Vis02a] Visarius, M. and Lessmann, J. and Kelso, F. and Hardt, W. and Amelunxen, C. and Scholand, A. and Ihmor, S. Definition of the IPQ Format with Respect to Specialized Description Features. Technical Report TR-IPL-2002-04, University of Paderborn, Informatik- und Prozesslabor, Warburger Strasse 100, 33098 Paderborn, Germany, August 2002.

[Vis02b] Visarius, Markus and Hardt, Wolfram and Ihmor, Stefan and Lessmann, Johannes. IPQ Format: Concepts and Implementation. In *Proc. of the Int. Workshop of the MEDEA+ Project A-511 ToolIP*, Madrid, Spain, January 2002.

[W3C02] W3C. *Extensible Markup Language (XML) 1.0 - Deutsche Übersetzung*, volume 2. Auflage. http://edition-w3c.de/TR/2000/REC-xml-20001006, 20.Januar 2002.

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