Beiträge zur Automatisierung der frühen Entwurfsphasen verteilter Systeme

DISSERTATION

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

eingereicht an der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von Dipl.-Inf. Tommy Baumann
geboren am 14.02.1977 in Neuhaus am Rennweg

Gutachter:
1. Prof. Dr. (PhD) Horst Salzwedel, TU Ilmenau
2. Prof. Dr.-Ing. habil. Wolfgang Fengler, TU Ilmenau
3. Prof. Dr.-Ing. Klaus D. Müller-Glaser, Univ. Karlsruhe (TH)

Tag der wissenschaftlichen Aussprache: 24.06.2009

urn:nbn:de:gbv:ilm1-2009000122

Zusammenfassung

Mit der rapide steigenden Geschwindigkeit elektronischer Bauelemente können Systeme mit erhöhter Komplexität, Vernetzung und Heterogenität entwickelt werden. Dies hat zur Folge, dass eine Entwicklung nur durch Teams von Spezialisten durchführbar ist. Gleichzeitig muss die Entwicklung parallel erfolgen, um eine möglichst frühzeitige Produkteinführung zu ermöglichen. Im traditionellen Entwurfsprozess wird daher der Entwurf in Form einer geschriebenen Spezifikation des Gesamtsystems erfasst und anschließend auf mehrere Teams aufgeteilt. Dies erfolgt in den frühen Entwurfsphasen, welche durch eine hohe Unsicherheit über das Produkt gekennzeichnet sind. Dabei müssen bei der Entwicklung der Subsysteme Annahmen und Entscheidungen getroffen werden, ohne den Einfluss auf das Gesamtsystem abschätzen zu können. Kopplungseffekte werden weitestgehend ignoriert. Viele kritische, insbesondere durch Kopplungseffekte hervorgerufene Fehler, können folglich erst bei der Integration am Ende der Entwicklung entdeckt werden. Zudem ist eine Optimierung des Gesamtsystems nicht möglich, da kein Gesamtsystemmodell vorliegt. Der traditionelle Entwurfsprozess besitzt daher ein hohes Entwicklungsrisiko.

Beim Entwurfsansatz Mission Level Design wird nach dem Konzeptentwurf anstatt einer geschriebenen eine ausführbare Spezifikation des Gesamtsystems entwickelt. Diese wird gegen die Gesamtsystemanforderungen validiert und optimiert. Daraufhin wird die Spezifikation des gekoppelten Gesamtsystems an mehrere Teams zur Entwicklung der Subsysteme weitergegeben, welche dann wieder zu einem Gesamtsystem integriert werden. Integrationsprobleme werden so schon in den frühen Entwurfsphasen gelöst, was eine wesentliche Verringerung von Entwicklungszeit und -risiko bewirkt.

Um die Spezifikationsqualität und -geschwindigkeit bei der Entwicklung von Gesamtsystemmodellen zu erhöhen, werden im Rahmen der Arbeit standardisierte Methoden zur Beschreibung und Leistungsbewertung verteilter Systeme, sowie zum automatisierten Mapping von Funktion in Architektur entwickelt. Dies ermöglicht bereits in den frühen Entwurfsphasen eine Architekturoptimierung des Gesamtsystems. Zusätzlich werden Methoden zur Überführung des abstrakten Entwurfs in Implementationen entwickelt.

Schlüsselwörter:
Systementwurf, Automatisierung, Simulation, Architektur, Performance

Abstract

With the rapid increasing speed of electronic devices systems with higher complexity, interconnectedness and heterogeneity can be developed. The development of such systems can only be done by teams of specialists. At the same time the development needs to happen in parallel to ensure an early time to market. Therefore in the traditional design process the design is described in form of a written specification of the common system and partitioned to several teams. This takes place in early design stages at high product uncertainty. Sub system development assumptions and decisions are made without being able to evaluate the effect on the common system. Thus many critical errors, especially those, caused by coupling effects, are discovered during system integration at the end of the design process. Furthermore an optimization of the common system is not possible, because of the lack of a common system model. Hence the traditional design process is a high risk development process.

In the Mission Level Design approach, an executable specification of the common system instead of a written specification is developed after concept development. These is validated and optimized against the requirements of the common system. The such validated specification of the coupled system is then passed on to specialist teams for sub system development. The sub systems are then integrated. In this manner integration problems can be solved in the early design stages. Development time and risk can be reduced significantly.

To increase the specification quality and speed while developing common system models, in the present work, standardized methods for specification and performance evaluation of distributed systems and methods for automated mapping of function into architecture are developed. This allows architecture optimization of the common system in the early design stages. In addition, methods for transformation of the abstract design into implementations are developed.

Keywords:
System Design, Automation, Simulation, Architecture , Performance

Thesen

1. Aufgrund der stetig steigenden Systemkomplexität ist die Entscheidungsfindung und damit der Entwurfsschwerpunkt in die frühen Entwurfsphasen zu verlagern.

2. Durch Kopplungseffekte zwischen Subsystemen hervorgerufene Fehler können anhand eines integrierten Entwurf auf Gesamtsystemebene erkannt und gelöst werden.

3. Eine Bewertung und Optimierung des Gesamtsystems setzt dessen Erfassung in Form einer ausführbaren Gesamtsystemspezifikation voraus.

4. Zur automatisierten Iteration über Entwurfsalternativen ist ein ausführbarer Entwurfsprozess notwendig.

5. Ein ausführbarer Entwurfsprozess kann anhand von Aktivitäts- oder Blockflussdiagrammen beschrieben werden, wobei zwischen Simulations-, Steuerungs- und Generatorelementen zu unterscheiden ist.

6. Voraussetzung für eine gemeinsame Performance-Analyse, plattformspezifische Synthese und Iteration über Architekturalternativen ist eine standardisierte, ausführbare Architekturbeschreibung.

7. Eine standardisierte Architekturbeschreibung beruht auf determinierten Schnittstellen und ermöglicht den Austausch, sowie die Wiederverwendung von Architekturkomponenten.

8. Performance-Analysen können anhand von Queuing-Methoden oder probabilistischen Methoden durchgeführt werden, wobei die Ergebnisse vom gewählten Abstraktionsniveau abhängen.

9. Die Festlegung einer Ausführungs- und Kommunikationsarchitektur, sowie die Verteilung der Funktion auf diese, hat einen signifikanten Einfluss auf die spätere System-Performance.

10. Durch Modell-Generierung können Modelle unter Verwendung einer steuernden Sprache spezialisiert oder generalisiert werden, wodurch ein Übergang zwischen Abstraktionsebenen ermöglicht wird.

(Ort, Datum) (Unterschrift)

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalte der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zu Folge hat.

(Ort, Datum) (Unterschrift)

Danksagung

Diese Arbeit widme ich meinem viel zu früh verstorbenen Vater Helmut Baumann. Ohne seine Unterstützung wäre mir weder Studium noch Promotion möglich gewesen.

Besonderer Dank bezüglich der Umsetzung gilt meinem Betreuer Prof. Dr. (PhD) Horst Salzwedel. Er stand mir jederzeit als Diskussionspartner für fachliche sowie menschliche Probleme zur Verfügung. Des weiteren bedanke ich mich bei den Mitarbeitern des Fachgebiets System- und Steuerungstheorie der Technischen Universität Ilmenau. Allem voran Dr.-Ing. Volker Zerbe, Dipl.-Inf. Maik Hauguth und Dipl.-Ing. Alexander Pacholik. Dank gebührt auch meinem Betreuer während meines Auslandsaufenthalts an der Stanford University M.Sc. David Hipkins.

Große Unterstützung erfuhr ich zudem durch meine ehemaligen Studenten Dipl.-Inf. Michael Rath, Dipl.-Inf. David Grüner, Dipl.-Inf. Nils Fischer, Dipl.-Inf. Frank Lohse, Dipl.-Inf. Matthias Eck und Dipl.-Inf. Stefan Riehmer, sowie durch die Mitarbeiter der Firma Mission Level Design GmbH Prof. Dr.-Ing. Gunar Schorcht, Dipl.-Inf. Thomas Lohfelder und M.Sc. Florin Stan Cotti. Ohne sie wäre die Durchführung der Arbeit undenkbar gewesen.

Zu guter Letzt danke ich meiner Lebensgefährtin Cathleen und meiner Familie für die unendliche Geduld, das entgegengebrachte Interesse und die vielen kleinen Hilfeleistungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Symbolverzeichnis

ARP ..... Address Resolution Protocol

ASIC ..... Application Specific Integrated Circuit

ATA ..... Air Transport Association

BER ..... Bit Error Rate

CAD ..... Computer Aided Design

CDFG ..... Control Data Flow Graph

CPIOM ..... Core Processing Input Output Module

CPU ..... Central Processing Unit

CSMA-CD ... Carrier Sense Multiple Access - Collision Detection

CTDE ..... Continous Time Discrete Event

DDF ..... Dynamic Data Flow

DE ..... Discrete Event

DGPS ..... Differential GPS

ECU ..... Electronic Control Unit

EDA ..... Electronic Design Automation

EIA232 ..... Electronic Industries Alliance 232

ESL ..... Electronic System Level

FPGA ..... Field Programmable Gate Array

FSM ..... Finite State Machine

GPS ..... Global Positioning System

ICMP ..... Internet Control Message Protocol

IGMP ..... Internet Group Management Protocol

INS ..... Inertial Navigation System

IP ..... Internet Protocol

IP-Cores ..... Intellectual Property Cores

IPv4 ..... Internet Protocol Version 4

ISO ..... International Organization for Standardization

LAN ..... Local Area Network

LRM ..... Line Replaceable Module

LRU ..... Line Replaceable Unit

MAC ..... Media Access Control

NoC ..... Network on Chip

NPC ..... Non-Deterministic Polynomial-Time Complete

NRV ..... Negative Differential Pressure Relief Valve

NTP ..... Network Time Protocol

OCU ..... Outflow Valve Control and Sensor Unit

OFV ..... Outflow Valve

OMG ..... Object Management Group

ORV ..... Overpressure Relief Valve

OSEK ..... Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug

OSI ..... Open Systems Interconnection

PLD ..... Programmable Logic Device

PTcl ..... Ptolemy Tool Command Language

PTLang ..... Ptolemy Language

QoS ..... Quality of Service

RAM ..... Random Access Memory

RDS ..... Radio Data System

SDF ..... Synchronous Data Flow

SoC ..... System on Chip

SV ..... Savety Valve

SysML ..... System Modeling Language

TCP ..... Transmission Control Protocol

TSIP ..... Trimble Standard Interface Protocol

UML ..... Unified Modeling Language

USB ..... Universal Serial Bus

UTP ..... Unshielded Twisted Pair

VHDL ..... Very Highspeed Integrated Circuit Hardware Description Language

XML ..... Extensible Markup Language

XSLT ..... Extensible Stylesheet Language for Transformation

Kapitel 1

Einleitung

1.1 Problemstellung

Moderne technische Systeme bestehen aus einer Vielzahl verteilter Komponenten und zeichnen sich durch architekturelle Komplexität, dynamische Interaktion und komplexe interdisziplinäre Funktionalität aus. Während Flugzeuge wie Boeing 777 in den neunziger Jahren circa 100 vernetzte Electronic Control Units (ECU) enthielten, sind für aktuelle Modelle wie Boeing 787 und Airbus 380 über 1000 zu veranschlagen [Salzwedel 2004a]. Eine ähnliche Entwicklung ist in der Automobilindustrie zu beobachten. Fahrzeuge wie der 5er BMW enthalten mittlerweile mehr als 100 ECU's [Rath u. a. 2006]. Ursache hierfür sind die wachsenden technologischen Möglichkeiten im Bereich der Elektronik, wobei von einer Rate zwischen 50 und 60 % pro Jahr ausgegangen werden kann [Moore 1965]. Dies wiederum ermöglicht innovative Systementwicklungen, wie beispielsweise neuartige Sicherheits- und Komfortsysteme. Da die Effizienzsteigerungen von Entwurfsmethoden mit maximal 25 % pro Jahr zu veranschlagen sind [Sikora u. Drechsler 2002], spricht man seit Mitte der neunziger Jahre vom sogenannten *System Design Gap*. Dieser Effekt wird durch immer kürzere Systemlebenszyklen in Verbindung mit frühzeitigen Produkteinführungsphasen weiter verstärkt.

Umso höher Komplexität und Verteilungsgrad steigen, desto häufiger tritt beim Entwurf das gleiche Problem auf: Einzelne Subsysteme werden anhand geschriebener Spezifikationen entworfen und validiert, funktionieren im Einzeltest einwandfrei, sobald sie jedoch zu einem Gesamtsystem integriert werden, kommt es zu unvorhersehbaren Fehlern, zum Systemversagen oder Scheitern ganzer Projekte. Grund hierfür ist die frühzeitige Verteilung von

Entwurfsaufgaben auf Teams von Spezialisten unter einer hohen Produktunsicherheit und ohne dass die *Kopplungseffekte* zwischen den Subsystemen berücksichtigt werden. Traditionelle Entwurfsansätze, wie beispielsweise die Problem Frame Methode [Jackson 2001], erlauben die Berücksichtigung solcher Kopplungseffekte nicht und besitzen daher eine hohes Entwicklungsrisiko. Dies lässt sich unter anderem durch die in Abbildung 1.1 dargestellten Ergebnisse der ESPITI-Studie [Schienmann 2002] belegen. Dort wurde gezeigt, dass 60 % aller kritischen Entwurfsprobleme in den frühen Entwurfsphasen während der Systemspezifikation auftreten.

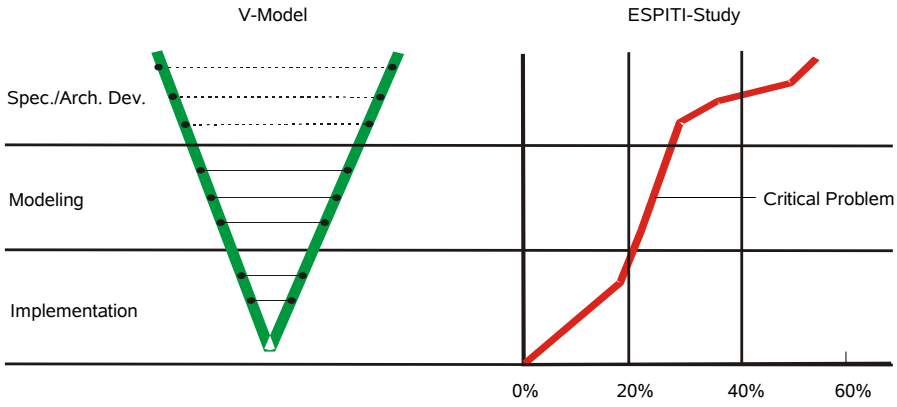

Abbildung 1.1: ESPITI-Studie [Salzwedel 2004a]

Beim Entwurfsansatz *Mission Level Design* [Schorcht 2000] wird eine *ausführbare Spezifikation* des Gesamtsystems entwickelt, welche gegen die Gesamtsystemanforderungen validiert und optimiert wird. Daraufhin wird die ausführbare Spezifikation des gekoppelten Gesamtsystems an mehrere Teams zur Entwicklung der Subsysteme weitergegeben, um diese im Anschluss wieder zu einem Gesamtsystem zu integrieren. Integrationsprobleme durch Kopplungseffekte können so frühzeitig erkannt und gelöst werden. Der Entwurfsschwerpunkt zur Analyse, Verifikation und Validierung liegt dabei in den *frühen Entwurfsphasen*. Nur so ist eine frühzeitige Entscheidungsfindung zur Vermeidung von Entwurfsproblemen möglich.

Aus dem verlagerten Entwurfsschwerpunkt und der Berücksichtigung der Gesamtsystemebene ergeben sich zwei grundlegende Probleme. Zum einen wird das Abstraktionsniveau erhöht und somit die Lücke zwischen abstraktem Entwurf und plattformspezifischer Implementation größer. Diese wird im Rah-

men der Arbeit als *System Abstraction Gap* bezeichnet. Eine manuelle Synthese ist aufgrund der Komplexität mit einem hohen Zeitaufwand verbunden. Zum anderen muss bereits in den frühen Entwurfsphasen eine *Entwurfsraumanalyse* (engl.: Design Space Exploration) zur Auffindung eines optimalen bzw. möglichen Entwurfs auf Gesamtsystemebene durchgeführt werden, was vor allem zur Ermittlung der Systemarchitektur von Interesse ist. Hierbei wird unter einer Entwurfsraumanalyse ein iterativer Prozess zur Bewertung einer endlichen Menge an Entwurfsvarianten bezüglich eines Entwurfsziels verstanden [Zentner 2006]. Zu beachten ist, dass im Sinne eines optimalen Gesamtentwurfs auch eine suboptimale Anforderungserfüllung durch Subsysteme möglich ist [Salzwedel 2007]. Manuelle Entwurfsraumanalysen sind infolge der Komplexität mit einem hohen Zeitaufwand verbunden.

1.2 Zielstellung

Ziel der Arbeit ist es, die *Spezifikationsqualität* und *-geschwindigkeit* beim Entwurf komplexer, verteilter Systeme in den frühen Entwurfsphasen zu steigern. Hierzu muss der Automatisierungsgrad während des Entwurfsprozesses erhöht werden. Das Mission Level Design unterstützt bereits einen modellbasierten Entwurf auf Gesamtsystemebene unter Berücksichtigung der Kopplungseffekte zwischen Subsystemen. Zudem liegen Systemspezifikationen unmittelbar in ausführbarer Form vor und können frühzeitig verifiziert und validiert werden. Da jedoch keine standardisierte Architekturbeschreibung existiert, fehlt die Grundlage zur automatisierten Durchführung iterativer Entwurfsraumanalysen in den frühen Entwurfsphasen. Eine Architekturoptimierung und Leistungsbewertung auf Gesamtsystemebene ist folglich mit einem hohen Aufwand verbunden. Auch fehlt ein gemeinsamer Ausgangspunkt zur Synthese.

Im Rahmen der Arbeit soll eine standardisierte, ausführbare Architekturbeschreibung entwickelt werden. Diese dient als Grundlage zur Definition von Architekturkomponenten, zum flexiblen Mapping von Funktion in Architektur, zur vereinheitlichten Leistungsbewertung, sowie zur automatisierten Erzeugung von Architekturmodellen [Baumann u. a. 2007a]. Dabei soll gezeigt werden, dass bereits in den frühen Entwurfsphasen eine automatisierte Iteration über Architekturalternativen möglich ist. Zudem sollen Grundlagen zur Definition eines ausführbaren Entwurfsprozesses untersucht werden, was Voraussetzung zur Steuerung der Architekturiteration ist. Schwerpunkt bildet in diesem Zusammenhang die Durchführung frühzeitiger Performance-Analysen,

durch welche eine ressourcenabhängige Leistungsbewertung dynamischer Interaktionen zwischen Subsystemen zur Auffindung geeigneter Architekturen ermöglicht wird. Relevante Performance-Eigenschaften sind beispielsweise Datendurchsatz, Speicherverbrauch oder Ausführungszeit auf einer Central Processing Unit (CPU). Zusätzlich sollen Transformationsvorschriften zur Synthetisierung von Implementationen auf Basis der standardisierten Architekturbeschreibung entwickelt werden.

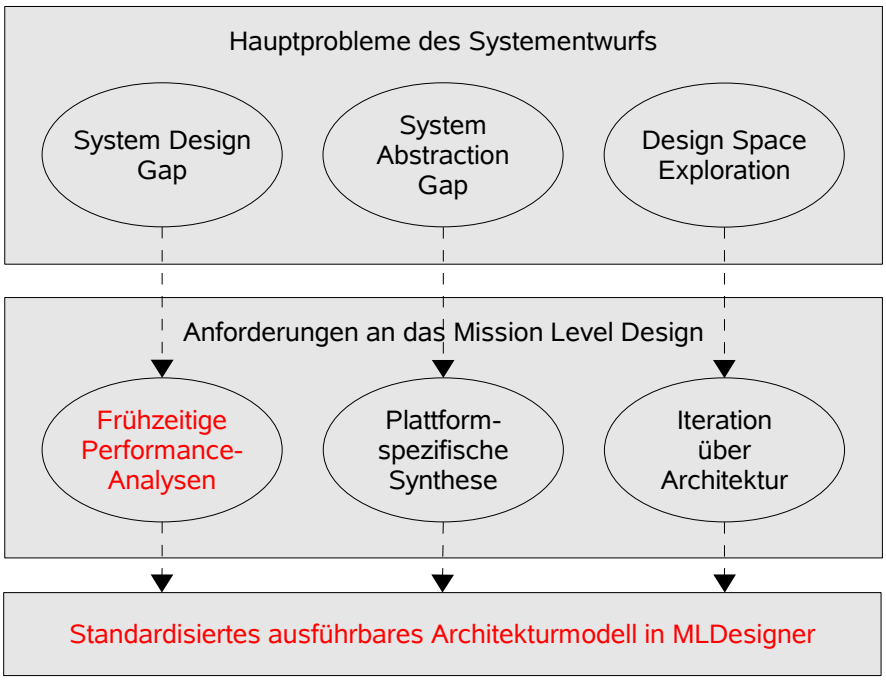

Abbildung 1.2: Ableitung der Zielstellung

Abbildung 1.2 fasst die Notwendigkeit zur Einführung eines *standardisierten ausführbaren Architekturmodells* zusammen. Dabei dienen die im vorhergehenden Abschnitt benannten Entwurfsprobleme und deren Interpretation als Anforderungen an das Mission Level Design als Ausgangspunkt. Zur Erfüllung der Zielstellung wird eine Erweiterung des Entwurfsansatzes Mission Level Design definiert und umgesetzt. Wie später noch gezeigt wird, beruht die Umsetzung auf einer Anwendungsumgebung für das Computer Aided Design (CAD) Tool bzw. Systementwurfswerkzeug *MLDesigner* [MLDe 2008]. Die Anwendungsumgebung wird dabei anhand mehrerer Beispiele validiert. Un-

ter anderem wird eine Architekturoptimierung für eine Avionik-Architektur in den frühen Entwurfsphasen durchgeführt. Ein weiteres Beispiel beschäftigt sich mit dem Entwurf eines Systems zur Positionsbestimmung und dessen Synthetisierung.

1.3 Gliederung

Die Arbeit gliedert sich in mehrere, thematisch aufeinander aufbauende Kapitel. Im einzelnen unterteilt sie sich wie folgt:

- **Kapitel 2 – Entwurfsansatzanalyse:**
  Ein einführendes Kapitel vermittelt zunächst Grundbegriffe des Systementwurfs, erläutert die Problemstellung der Arbeit, identifiziert Anforderungen an den Entwurf und analysiert den traditionellen Entwurf sowie den Entwurfsansatz Mission Level Design zur Ableitung von Lösungsvorschlägen.

- **Kapitel 3 – Automatisierung des Mission Level Designs:**
  Das dritte Kapitel wendet sich der Kernthematik der Arbeit zu. Dort wird der Entwurfsansatz Mission Level Design in Bezug auf die ermittelten Anforderungen erweitert, um eine Erhöhung des Automatisierungsgrads zu erreichen. Außerdem werden Schlussfolgerungen hinsichtlich der Umsetzung gezogen.

- **Kapitel 4 – MLDesigner-Anwendungsumgebung:**
  Auf Grundlage des erweiterten Mission Level Designs behandelt das vierte Kapitel dessen Umsetzung in Form einer Anwendungsumgebung für das Systementwurfswerkzeug MLDesigner. Hierbei wird zunächst auf MLDesigner und die Eigenheiten verteilter Systeme eingegangen. Im Anschluss wird die Anwendungsumgebung realisiert.

- **Kapitel 5 – Anwendungsbeispiele:**
  Das fünfte Kapitel beschreibt an praktischen Beispielen den Einsatz der entwickelten Anwendungsumgebung. Hierzu werden verschiedene Ausführungs- und Kommunikationskomponenten, sowie komplette Systemmodelle entworfen, simuliert und bewertet.

- **Kapitel 6 – Zusammenfassung:**
  Im sechsten und letzten Kapitel werden die Ergebnisse der Arbeit zusammengefasst und bewertet. Zudem wird auf Ansatzpunkte für weiterführende Arbeiten eingegangen.

Kapitel 2

Entwurfsansatzanalyse

2.1 Begriffe und Relationen

Innerhalb des vorliegenden Abschnitts werden zunächst grundlegende Begriffe aus dem Bereich Systementwurf eingeführt und deren Beziehungen untereinander erläutert. Dabei ist zu beachten, dass in der Literatur oftmals mehrere Definitionen für ein und den selben Begriff existieren. Als Basis für die weiteren Ausführungen soll ein System wie folgt definiert sein:

Definition 1: Ein *System* ist ein mit der Außenwelt kommunizierendes Objekt, das sich durch die Abgrenzung von seiner Umwelt bildet. Es besteht aus einer Anordnung von Komponenten, die miteinander in Beziehung stehen und gemeinsam die Systemfunktion definieren.

Im Bereich der Systemtheorie wird hierbei zwischen einem extern beobachtbaren Verhalten und einer internen Struktur differenziert [Zeigler u. a. 2000]. Unter externem Verhalten versteht man die Verbindung von Eingaben zu Ausgaben. Die interne Struktur beschreibt Zustandsübergänge bezüglich der Eingaben und des aktuellen Zustands, sowie Ausgaben anhand des aktuellen Zustands. Kenntnisse über die interne Struktur erlauben die Analyse und Ausführung von Systemen.

Real existierende Systeme bestehen meist aus einer Vielzahl kleinerer Subsysteme und besitzen bestimmte Eigenschaften, wie Zeitverhalten, Änderungsverhalten, Übertragungsverhalten und Reproduzierbarkeit [Adamski 2001]. Sie sind allgegenwärtig und in den unterschiedlichsten Bereichen anzutreffen, wie z.B. physikalische, biologische, chemische, ökonomische oder technische

Systeme. Bekannte Beispiele aus dem technischen Bereich sind Computernetzwerke, Handys, Ampelsteuerungen oder Antiblockiersysteme. Anzumerken ist, dass die meisten technischen Systeme wiederum als eingebettete Systeme zu klassifizieren sind.

Definition 2: Unter einem *eingebetteten System* versteht man eine elektronische Recheneinheit in Hard- und Software, die in einen technischen Kontext eingebunden ist und in Bezug auf diesen Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben erfüllt.

In diesem Zusammenhang spricht man von System on Chip (SoC), wenn alle elektronischen Komponenten auf einem Chip integriert sind. Als Network on Chip (NoC) bezeichnet man die Erweiterung von SoC durch ein internes Bussystem.

Systeme besitzen im Allgemeinen eine bestimmte Funktionalität bzw. müssen bestimmte Aufgaben erfüllen. Hieraus lassen sich eine Reihe von Anforderungen ableiten. Diese sind während der Systementwicklung zu berücksichtigen und möglichst vollständig umzusetzen.

Definition 3: *Anforderungen* sind die Definitionen der geforderten Eigenschaften eines Systems.

Eine weit verbreitete Möglichkeit zur Klassifikation ist die Unterteilung in *funktionale Anforderungen* und *nichtfunktionale Anforderungen* [Glinz 2005]. Erstere beziehen sich auf die Aufgaben des Systems und die damit verbundene Informationsverarbeitung, das heißt auf Daten, Operationen und Verhalten. Letztere auf die Eigenschaften des Systems bzw. auf die Umstände zur Erbringung der geforderten Funktionalität, wobei man zwischen Leistungsanforderungen, Qualitätseigenschaften und weiteren Randbedingungen (engl.: Constraints) unterscheidet.

Ausgehend von den erfassten Anforderungen können zu entwerfende bzw. existierende Systeme in Spezifikationen überführt werden. Jede Spezifikation beinhaltet eine Auflistung aller Komponenten und Subsysteme inklusive ihrer Kennwerte, sowie Informationen über gewünschte bzw. vorhandene Kollaborationsbeziehungen.

Definition 4: Die *Spezifikation* ist eine formale Systembeschreibung für den Entwurf. Sie definiert und quantifiziert Eigenschaften eines Systems.

Um anhand einer Spezifikation Systemeigenschaften zu untersuchen oder ein real existierendes System abzuleiten, müssen anwendungsspezifische Ent-

wurfsprozesse durchlaufen werden. Diese definieren Konzepte und Regeln zur schrittweisen Entwicklung von Systemen. Mehrere solcher Ansätze werden in Kapitel 2 erläutert.

Definition 5: Ein *Design Flow* (Entwurfsprozess, Entwurfsansatz, Entwurfsmethodik) ist die iterative Transformation einer Spezifikation funktionaler und/oder nicht-funktionaler Natur auf die Elemente und Relationen des Zielsystems.

Design Flows werden meist durch sogenannte Systementwurfswerkzeuge bzw. CAD-Tools umgesetzt, wobei speziell bei elektronischen Systemen auch die Bezeichnung EDA-Tools, was für Electronic Design Automation (EDA) steht, verwendet wird. Hierbei ist es vorteilhaft Spezifikationen im Vorab durch Modelle nachzubilden. Modelle erlauben eine kostengünstige und beschleunigte Untersuchung relevanter Systemeigenschaften.

Definition 6: Ein *Modell* ist eine vereinfachte Nachbildung eines existierenden oder gedachten Systems mit seinen Prozessen in einem anderen begrifflichen oder gegenständlichen System. Es unterscheidet sich hinsichtlich der untersuchungsrelevanten Eigenschaften nur innerhalb eines vom Untersuchungsziel abhängigen Toleranzrahmens vom Vorbild [Ingenieure Nov. 1996].

Demzufolge bezieht sich ein Modell stets auf eine Teilmenge von Eigenschaften des realen Systemvorbilds, wird in Bezug auf diese entworfen und liefert auch nur in Bezug auf diese verwendbare Ergebnisse. Beispielsweise ist es ein Unterschied, ob einerseits die Timing-Eigenschaften eines Systems im Vordergrund stehen oder andererseits das Systemverhalten idealisiert, sprich zeitlos, betrachtet wird. Voraussetzung zur Erstellung eines Modells ist die Definition und Verwendung einer formalen Sprache, idealerweise in Verbindung mit einer grafischen Notation. Da eine schier unüberschaubare Menge solcher Sprachen existiert, sei an dieser Stelle die von der Object Management Group (OMG) entwickelte Unified Modeling Language (UML) [OMG 2004] erwähnt, durch welche eine standardisierte Sprache zur Beschreibung von Systemstruktur und -verhalten definiert wird. Die meisten Systementwurfswerkzeuge verwenden dennoch ihre eigenen Design-Flow-spezifischen Modellierungsansätze, obgleich gewisse Analogien zur UML-Spezifikation oftmals unverkennbar sind.

Weit verbreitet sind *blockorientierte Ansätze*, beispielsweise zu finden in ML-Designer (siehe 4.2), Ptolemy II [Univ 2005] und Rational Rose Realtime [IBM 2003]. Ein Block ist als eine Art von White Box zu verstehen, welche

über eine Ein- und Ausgabeschnittstelle verfügt, eine beliebig beschreibbare Funktion umsetzt und Parameter zur Anpassung an den aktuellen Anwendungsfall besitzt. Einmal definierte Blöcke können innerhalb hierarchischer Blockflussdiagramme mehrfach wiederverwendet und zueinander in Beziehung gesetzt werden. Dies erlaubt die Dekomposition von Systemverhalten zur Komplexitätsreduktion und zur Aufgabenverteilung auf Entwicklergruppen. Des weiteren folgt aus der hierarchischen Strukturierung die Möglichkeit unterschiedliche Ausführungsmodelle zuzuordnen, durch welche die Systemdynamik, das heißt das zeitliche Systemverhalten von Prozessen, determiniert werden kann.

Solche Prozesse können diskret oder kontinuierlich sein, wobei die zweitgenannten mit Hilfe von Zeit- und Werterastern diskretisierbar sind. Diskrete Prozesse lassen sich auf Basis einer Taktung oder anhand auftretender Ereignisse digital steuern. Kommt eine Taktung zum Einsatz, spricht man von *zeitgesteuerten Prozessen*. Wird im Gegensatz dazu auf Ereignisse reagiert, handelt es sich um *ereignisgesteuerte Prozesse*. Erstere können durch synchrone und letztere durch asynchrone Automaten beschrieben werden. Daraus folgt, dass die Systemdynamik durch unterschiedliche Ausführungsmodelle bzw. Domänen erfasst werden kann. Eine umfassende Beschreibung hierzu findet man unter [Lee 2002]. Die dort genannten Konzepte beziehen sich ursprünglich auf die funktionale Ebene, sind aber grundsätzlich unabhängig vom verwendeten Abstraktionsgrad und werden im Bereich des Systementwurfs universell eingesetzt.

Definition 7: Eine *Domäne* definiert die allgemeine Interaktionssemantik in Kollaboration stehender Modellkomponenten, jedoch nicht deren Ausführung.

Um aus Modellen einen Erkenntnisgewinn abzuleiten, welcher auf das reale bzw. zu entwerfende System bezogen werden kann, müssen diese formal oder informal analysiert werden. Dabei sind formale Analysen infolge der erhöhten Systemkomplexität aktueller technischer Systeme kaum mehr durchführbar. Aus diesem Grund ist es notwendig modellbasierte Simulationstechniken einzusetzen, sprich Systeme in Form von *ausführbaren Systemspezifikationen* zu entwerfen.

Definition 8: Die *Simulation* ist ein Verfahren zur Nachbildung eines Systems mit seinen dynamischen Prozessen in einem experimentierbaren Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind [Ingenieure Nov. 1996].

Ein Prozess ist in diesem Zusammenhang als Transport-, Speicherungs- oder Umformungsvorgang von Energie, Materie oder Information zu verstehen, wobei die Dynamik der Prozesse durch den Begriff des Zustands qualifiziert werden, der die zeitliche Veränderung interner Systemgrößen determiniert. Kommt es zur Simulation, wird basierend auf der jeweiligen Verhaltens- und Strukturbeschreibung des Modells ein domänenspezifischer Datenfluss erzeugt, welcher im Anschluss oder während der Simulation analysiert werden kann. Hierdurch wird der Entwickler während des Entwurfs unterstützt [Bennett 1995] und ein besseres Verständnis komplizierter dynamischer Prozesse erreicht [Adamski 2001]. Ziel einer jeden Simulation ist es Rückschlüsse auf das reale bzw. zu entwerfende System zu ziehen, wobei sich die Ergebnisse stets auf die im Modell erfassten untersuchungsrelevanten Systemeigenschaften beziehen und nicht allgemeingültig sind.

Basierend auf der Systemausführung und der damit verbundenen simulativen Analyse kann die Systemspezifikation und deren Komponenten gegen die gestellten funktionalen und nichtfunktionalen Anforderungen getestet werden. Ein solcher Vorgang wird üblicherweise als Validierung bezeichnet. Dabei wird getestet, ob das nachgebildete Verhalten dem erwarteten entspricht. Prinzipiell unterscheidet man bei der Validierung zwischen einer formalen und simulativen Variante, wobei die erstgenannte für komplexe Systemspezifikationen nahezu unmöglich ist [Salzwedel 2004a]. Ursache hierfür ist der exponentielle Anstieg der zu überprüfenden internen Zustände bezüglich der steigenden Systemkomplexität. Beispiel für eine zu validierenden Anforderung ist unter anderem die Einhaltung einer definierter Antwortzeit bezüglich bestimmter Systemeingaben.

Definition 9: *Validierung* ist die Beweisführung, dass eine Systemspezifikation die gestellten Anforderungen erfüllt. Die umgangssprachliche Formulierung lautet: „Bauen wir das richtige Produkt?“.

Im Gegensatz zur Validierung dient die Verifikation zur Überprüfung der formalen Korrektheit des Modells bzw. der Systemspezifikation. Beispielsweise wird überprüft, ob ein bestimmter Syntax eingehalten wird oder die verwendeten Modellkomponenten korrekt verbunden sind. Moderne Systementwurfswerkzeuge verifizieren automatisiert während des Entwurfs und vor der Durchführung von Simulationen.

Definition 10: Unter *Verifikation* versteht man den formalen oder informalen Beweis, dass eine konkrete Implementation der vorgegebenen Spezifikation entspricht. Die umgangssprachliche Formulierung lautet: „Bauen wir das Produkt richtig?“.

Ein weiterer wichtiger Begriff aus dem Bereich Systementwurf ist die sogenannte Synthese. Ziel ist es, eine formale und möglichst validierte Systemspezifikation in eine konkretisierte Implementation zu überführen. Hierbei bezieht sich jede Synthese auf genau zwei Abstraktionsebenen und die Transformation zwischen diesen. Da ein Übergang von einer abstrakteren zu einer spezielleren Ebene stattfindet, ist die Synthese durch das Fehlen steuernder und implementationsspezifischer Details geprägt.

Definition 11: Als *Synthese* wird möglichst automatisierte Transformation zwischen Abstraktionsebenen und Sichten bezeichnet.

Ein mögliches Beispiel stellt die Überführung einer auf abstrakter Systemebene durch Boolesche Gleichungen oder Zustandsmaschinen beschriebenen Spezifikation zur Logikebene mit Gattern, Registern und Netzlisten dar. Gleichzeitig ist eine Transformation von Systemebene zur Algorithmenebene in C-Code oder der Very Highspeed Integrated Circuit Hardware Description Language (VHDL) durchführbar [Rath u. Salzwedel 2004]. Anzumerken ist, dass eine Transformation oftmals schwierig oder unmöglich ist, wenn die ausgehende Beschreibungssprache dem Entwickler zu viele Freiheiten lässt. Beispielsweise resultiert aus der dynamischen Speicherverwaltung und Polymorphie der auf C++ basierenden SystemC-Bibliothek eine nichtdeterministische Objektanzahl und -typisierung zur Compilezeit und damit auch während der Durchführung einer Synthetisierung. Aktuelle Ansätze versuchen diese Problematik durch die Verwendung von Scope-Tabellen zu lösen [Grimpe u. Oppenheimer 2003].

Voraussetzung zur Durchführung einer Synthese ist die Verteilung der Systemfunktionen auf eine zuvor festgelegte Architektur. In diesem Zusammenhang spricht man auch von einer Partitionierung. Dabei ist zu beachten, dass der Begriff Partitionierung mehrfach belegt ist und sich hier lediglich auf die Festlegung einer Architektur bezieht.

Definition 12: Unter *Partitionierung* versteht man die Aufteilung der Systemfunktionalität anhand definierter Entwurfsbeschränkungen auf eine plattformspezifische Architektur.

Der Partitionierungsprozess besteht aus zwei Phasen, die als Allokation und

Bindung bezeichnet werden [Buchenrieder 1999]. Während der Allokation wird eine Ausführungs- und Kommunikationsarchitektur festgelegt, während der Bindung die Funktion unter Beachtung von Randbedingungen auf diese verteilt. Solche Randbedingungen können beispielsweise Kostenbeschränkungen, Leistungsaufnahmen oder Durchsatzraten sein. In diesem Zusammenhang gestaltet sich die Definition einer allgemeingültigen Kostenfunktion als schwierig, da weitere Entscheidungskriterien, wie geschätzte Lebensdauer, geplante Verkaufszahlen, Gesamtproduktionskosten, Komponentenwiederverwendung, Lagerkosten, Modulvielfalt, Servicefreundlichkeit und Komponentenanzahl relevant sind. Auch die algorithmische Umsetzung ist als problematisch einzustufen, da die Partitionierung eine spezielle Form des allgemeinen Partitionierungsproblems darstellt, welches zur Komplexitätsklasse Non-Deterministic Polynomial-Time Complete (NPC) gehört.

Nach einer erfolgreichen Partitionierung mit anschließender Synthese liegt die Systemspezifikation in Form eines Prototypen vor, der aus mehreren plattformspezifischen Hard- und Softwarekomponenten besteht. Diese müssen zur Validierung wiederum gegen die Systemanforderungen getestet werden.

2.2 Anforderungen an den Entwurf

Mit der steigenden Komplexität, Vernetzung und Heterogenität zu entwickelnder Systeme steigen auch die Anforderungen an den Entwurfsprozess. Im weiteren Verlauf werden diese auf Grundlage der Problemstellung in Abschnitt 1.1 abgeleitet und erläutert.

Um ein System entwerfen zu können, muss eine *Anforderungsanalyse* durchgeführt werden. Darunter versteht man die Ermittlung und Spezifikation der Anforderungen des Auftraggebers bzw. Nutzers an das zu entwickelnde System. Dabei muss die Erfassung formal erfolgen, um eine simulative Validierung der Beziehungen zwischen den Anforderungen zu ermöglichen.

Durch Verschiebung des *Entwurfsschwerpunkts in die Spezifikationsphase* können Systeme frühzeitig validiert, analysiert und bewertet werden. Dies ermöglicht eine frühzeitige Entscheidungsfindung, sowie eine frühzeitige Auffindung von Entwurfsfehlern. Kostenintensive Folgefehler können so bereits während der Systemspezifikation vermieden werden [Salzwedel u. a. 2008]. Die in Abbildung 1.1 dargestellten Ergebnisse der ESPITI-Studie verdeutlichen diesen Zusammenhang genauer. Dort wird gezeigt, dass die meisten kritischen Entwurfsprobleme in der Spezifikationsphase auftreten, da hier die Unsicherheit

über das zu entwerfende System bzw. Produkt am höchsten ist. Folglich muss der Entwurfsschwerpunkt in der Spezifikationsphase liegen, um das Entwicklungsrisiko so weit wie möglich zu reduzieren. Diese Annahme wird zusätzlich durch den äquivalenten Verlauf der Produktunsicherheits-Kurve aus Abbildung 2.6 und Entwurfsproblem-Kurve aus Abbildung 1.1 gestützt. Dabei bezieht sich der Vergleich lediglich auf den Verlauf bis zur Produktfreigabe, da die ESPITI-Studie auf den Entwurfsprozess fokussiert.

Bedingung zur Verschiebung des Entwurfsschwerpunkts ist die Erfassung von Funktion, Architektur und Umgebung bzw. Mission des Systems in Form eines ausführbaren Gesamtsystemmodells [Salzwedel 2004b]. Nur so kann bereits in der Spezifikationsphase der Entwurf simuliert und gegen die Missionen getestet werden. Beispielsweise können frühzeitige Performance-Analysen zur Bewertung unterschiedlicher Architekturalternativen durchgeführt werden [Baumann u. a. 2007a]. Der *Entwurf auf Gesamtsystemebene* erlaubt somit die Analyse, Verifikation und Validierung des zu entwerfenden Systems am virtuellen Prototypen.

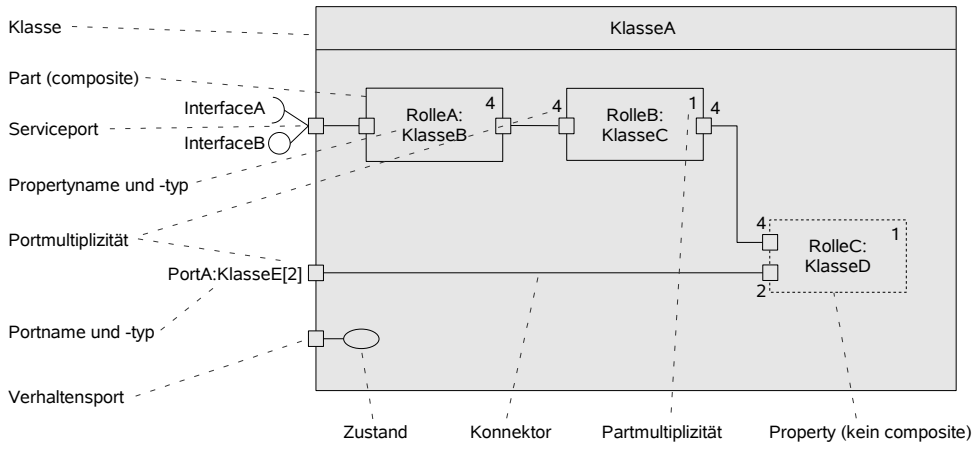

Abbildung 2.1: Kompositionsstrukturdiagramm

Um Analysen am Gesamtsystem durchführen zu können, muss dieses zunächst in Form eines Modells erfasst werden. Modelle sind als natürliches Mittel zur Erfassung und Abstraktion von Systemen zu verstehen und ermöglichen eine rechentechnisch weiterverarbeitbare Beschreibung von Systemanforderungen und Entwurfskriterien. Zudem bilden sie die Grundlage zur Datenhaltung, Austauschbarkeit und Wiederverwendbarkeit. Werden Modelle im

Bereich des Systementwurfs eingesetzt, spricht man auch von einem *modellbasierten Entwurf*. Dabei definieren die Systemanforderungen was ein Modell tun soll, und bilden gleichzeitig die Grundlage zur Festlegung des erforderlichen Abstraktionsniveaus. Beispiele für Modellierungssprachen sind unter anderem SystemC [Grötker u. a. 2003] oder die in MLDesigner [MLDe 2008] und Rational Rose Realtime [IBM 2003] verwendete Kombination von Blockflussdiagrammen bzw. Kompositionsstrukturdiagrammen und Zustandsmaschinendiagrammen. Letztere werden in den Abbildungen 2.1 sowie 2.2 dargestellt und sind grundlegender Bestandteil der UML sowie System Modeling Language (SysML) [Vanderperren u. Dehaene 2005].

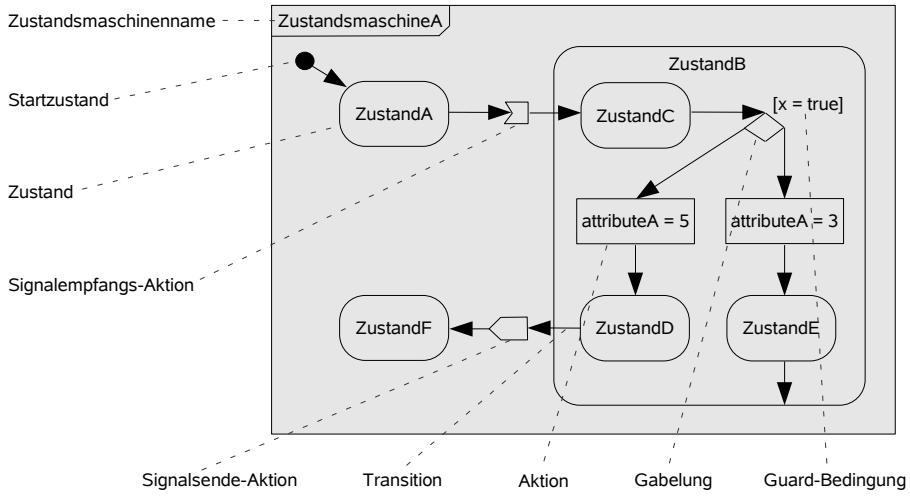

Abbildung 2.2: Zustandsmaschinendiagramm

Eine weitere Voraussetzung bildet die modellbasierte Spezifikation von Systemen in Form von *ausführbaren Spezifikationen*. Hierdurch kann der Entwurf im Gegensatz zu geschriebenen Spezifikationen simulativ analysiert, verifiziert [Pacholik u. Fengler 2007] und gegen die Systemanforderungen validiert werden. Modellunsicherheiten und -fehler können beschleunigt aufgefunden, dynamische Ressourcenauslastungen bemessen und die Einhaltung von Randbedingungen überprüft werden [Salzwedel u. a. 2008]. Dabei ist zu beachten, dass eine zeiteffiziente Verifikation mit rein formalen Methoden infolge des Komplexitätsanstiegs nicht länger durchführbar ist, weswegen bei aktuellen Ansätzen formal beschriebene Modelleigenschaften während der Simulation überprüft werden [Jesser u. a. 2007]. Ausgangspunkt zur Erstellung

einer ausführbaren Spezifikation bildet der Entwicklung eines Modells, welches ein Datenaustauschkonzept unterstützen muss, sowie die Auswahl geeigneter Ausführungsmodelle bzw. -domänen zur Kontrolle des Datenflusses. Es existiert eine Vielzahl solcher Ausführungsmodelle, welche unter anderem in [Lee 2002] und Kapitel 4.2 erläutert werden. Beispielsweise lassen sich physikalische Prozesse durch kontinuierliche, zeitorientierte Abläufe durch ereignisgesteuerte und synchrone Abläufe durch datenflussorientierte Ausführungsmodelle beschreiben.

Da Gesamtsystemmodelle unterschiedliche Prozesse und Abläufe zur Abbildung von Funktion, Architektur und Umwelt bzw. Mission beinhalten, können diese präziser durch mehrere, in Interaktion stehende Ausführungsmodelle erfasst werden. Ein solch kombinierter Einsatz wird als *Multidomänen-Entwurf* bezeichnet. Ein Beispiel für die kombinierte Verwendung wird in [Baumann u. a. 2008] beschrieben. Dort wird unter anderem das Verhalten eines Satelliten zum experimentellen Nachweis der Relativitätstheorie durch ein kontinuierliches und die Funktion sowie Architektur der Satellitensteuerung durch ein ereignisgesteuertes Ausführungsmodell realisiert.

Um die Modellerstellung und -ausführung zu beschleunigen, muss die Komplexität der Modelle so weit wie möglich gesenkt werden [Pidd 1997]. Dies wird durch die Abstraktion separater, analyserelevanter Systemeigenschaften erreicht. Dahinter verbirgt sich die Idee, dass Modelle nur so detailliert zu entwerfen sind, dass die an sie gerichteten Fragestellungen mit einer ausreichenden Güte beantwortet werden können. Aus diesem Grund müssen Entwurfsansätze ein *variables Abstraktionsniveau* unterstützen. Nur so können verschiedene zweckgebundene Sichten auf das zu entwerfende System erstellt und analysiert werden. Beispiele für den Entwurf mit unterschiedlichen Abstraktionsniveaus sind die separate Betrachtung von Funktion, Durchsatzrate, Ausfallsicherheit oder Leistungsaufnahme.

Aus der steigenden Systemkomplexität folgt die Notwendigkeit zur Verteilung der Entwurfsaufgaben auf Teams von Spezialisten. Dabei können die spezifizierten Subsysteme aufgrund von Kopplungseffekten nicht separat entwickelt werden. Ein Beispiel für solche Kopplungseffekte sind dynamische Kommunikationsverzögerungen durch parallele Subsystem-Zugriffe. Demzufolge ist ein *integrierter Entwurf* notwendig. Konkret bedeutet dies, dass vor der Verteilung bzw. Dekomposition der Subsystem-Spezifikationen auf Teams von Spezialisten, eine Gesamtsystemspezifikation erstellt wird, welche zur Analyse der Kopplungseffekte und zur Validierung dient. Erst anschließend werden die Subsystem-Spezifikationen konkretisiert, verteilt, entworfen und wieder zu einem Gesamtsystem integriert. Eine nachgelagerte, fehleranfällige Integration

wird vermieden, wodurch die Unsicherheit über das zu entwerfende System bzw. Produkt frühzeitig verringert werden kann [Salzwedel u. a. 2009].

Eine weitere sich aus der Verschiebung des Entwurfsschwerpunkts ergebende Anforderung ist die Unterstützung einer frühzeitigen *Optimierung auf Gesamtsystemebene*. Ziel ist es, die verfügbaren Ressourcen so zu verteilen, dass die Anforderungen an das Gesamtsystem optimal erfüllt werden [Fischer 2007]. Abbildung 2.3 zeigt beispielhaft den Entwurfsraum zweier Subsysteme X und Y bezogen auf zwei Gesamtsystemanforderungen A und B. Jedes Subsystem hat eine untere und oberen Schranke zur Erfüllung der

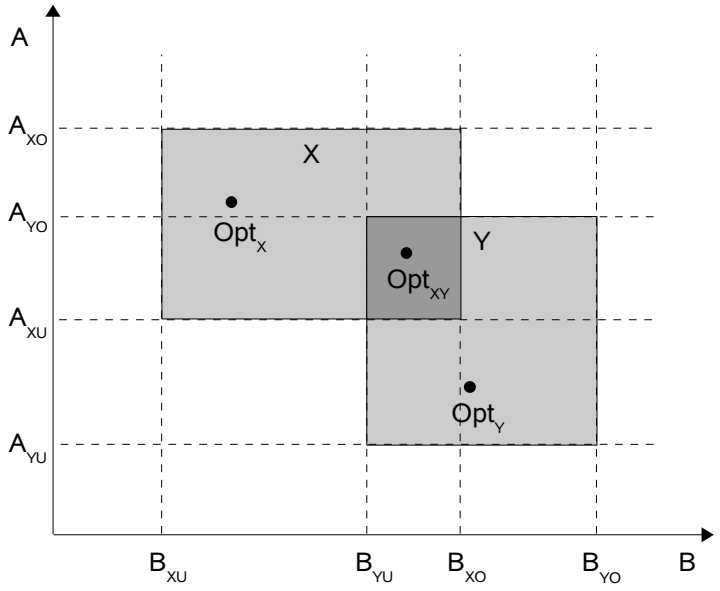

Abbildung 2.3: Gesamtsystemanforderungen in Relation

Anforderungen A_{YU}, A_{XU}, A_{YO}, A_{XO}, B_{XU}, B_{YU}, B_{XO}, B_{YO}, sowie ein Optimum Opt_X und Opt_Y. Im Rahmen des Entwurfs müssen die Ressourcen so verteilt werden, dass die Anforderungen optimal erfüllt werden, was dem geschnittenen Bereich bzw. dem Punkt Opt_{XY} entspricht. In diesem Zusammenhang spricht man von einem pareto-optimalen Entwurf, wenn alle Ressourcen zugeordnet wurden und keine weitere Änderung bezüglich eines Subsystems eine optimalere Erfüllung bringt, ohne dass mindestens ein anderes Subsystem schlechter gestellt wird [Givargis u. a. 2002]. Im Anschluss an die Optimierung auf Gesamtsystemebene dient die ermittelte Ressourcenzuord-

nung als Ausgangspunkt zur Konkretisierung der Subsystem-Spezifikationen. Hierduch kann die Unsicherheit des Entwurfs frühzeitig gesenkt werden. Voraussetzung zur Durchführung einer Optimierung ist die iterative Erzeugung und Bewertung von Entwurfsalternativen.

Um die Iteration über verschiedene Entwurfsalternativen zu beschleunigen, muss der Entwurfsprozess in Form eines *ausführbaren Entwurfsprozesses* automatisiert werden. Darunter versteht man eine ausführbare Spezifikation, welche eine schrittweise bzw. iterative Konkretisierung, Analyse und Optimierung eines Entwurfs hinsichtlich anforderungsspezifischer Kriterien durch kontrollierte Ausführung in Relation stehender Entwurfsschritte ermöglicht. Vereinfacht ausgedrückt handelt es sich um eine Simulation von Simulationen, wobei die externe Simulation durch die internen beeinflusst bzw. gesteuert werden kann. Als wesentliche Vorteile eines ausführbaren Entwurfsprozesses sind die Möglichkeit zur Automatisierung der Entwurfsraumanalyse, sowie zur simulativen Validierung des Entwurfsprozesses selbst zu nennen. Dabei verfolgt die Entwurfsraumanalyse die Zielstellung, sowohl das Verhalten als auch die Struktur von Systemmodellen bzw. Gesamtsystemmodellen in Bezug auf die gestellten funktionalen und nichtfunktionalen Anforderungen zu bestimmen.

Voraussetzung für die Verwendung eines ausführbaren Entwurfsprozesses ist die Möglichkeit zur *dynamischen Modellgenerierung*. Nur so können Entwurfsalternativen mit unterschiedlicher Struktur und Parametrisierung durch einen steuernden Prozess erzeugt werden. Grundlage bildet zum einen die modellbasierte Erfassung von Systemeigenschaften und zum anderen die Verwendung einer Sprache zur Beschreibung und Konfiguration zu generierender Modelle [Rath 2007].

Aufgrund der Verschiebung des Entwurfsschwerpunkts in die Spezifikationsphase, wird die Zuordnung von Funktion auf Architektur in Abhängigkeit der verfügbaren Ressourcen bereits dort durchgeführt. Demzufolge muss eine *Synthese auf Basis des Gesamtsystems* unterstützt werden, um den abstrakten Entwurf in plattformspezifische Implementationen zu überführen. Voraussetzung ist ein Standard zur Definition von Zielplattformen bzw. Architekturmodellen [Sangiovanni-Vincentelli u. Martin 2001]. Zu beachten ist, dass abstrakte Modelle durch ein Fehlen implementationsspezifischer Details gekennzeichnet sind, weswegen diese im Vorab zu ergänzen sind.

2.3 Entwurfsansätze

2.3.1 Traditioneller Entwurf

Im Rahmen des Abschnitts werden traditionelle Techniken für den methodischen Entwurf von Systemen erläutert. Dabei werden mehrere Entwurfsansätze aufgeführt und aufeinander bezogen.

Ein bekannter Vertreter ist das relativ einfach strukturierte *Wasserfallmodell* [Royce 1987]. Es besteht aus mehreren sukzessiv angeordneten Entwicklungsschritten:

- Analyse der Anforderungen
- Entwurf des Systems
- Implementierung des Systems
- Durchführung von Tests
- Einsatz des Systems

Da Test und Validierung erst am Ende durchgeführt werden, können Fehler in vorhergehenden Entwicklungsschritten nahezu unmöglich erkannt und behoben werden. Somit wird der Raum für Entwurfsiterationen eingeschränkt, was in einem suboptimalen System resultiert. Weiterhin ist eine strenge Abarbeitung und Trennung einzelner Projektphasen in der Praxis nur schwer umzusetzen. Dem Wasserfallmodell lässt sich dennoch die Erkenntnis abgewinnen, dass Systeme in mehreren in sich geschlossenen Schritten entworfen werden können.

Weitere Entwurfsstrategien versuchen die Probleme des Wasserfallmodells zu minimieren, wie beispielsweise das in Abbildung 2.4 dargestellte *Spiralmodell* [Boehm 1988]. Die Spirale beginnt im Ursprung eines Koordinatensystems, dessen Quadranten die folgenden vier Phasen darstellen:

- Bestimmung von Zielen, Alternativen und Begrenzungen
- Einschätzung der Alternativen und Identifizierung von Risiken
- Entwurf und Verifikation
- Planung der nächsten Phase

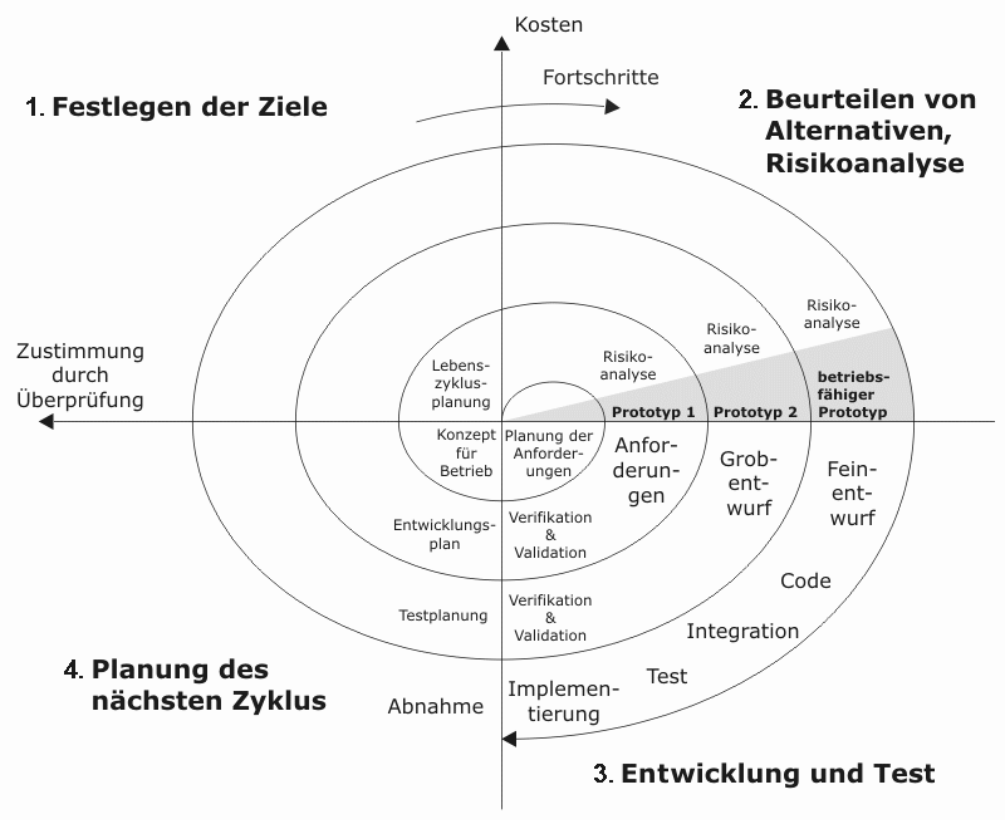

Abbildung 2.4: Spiralmodell [Boehm 1988]

Die Phasen werden sukzessiv durchlaufen, wobei diese in vier Durchläufe eingebettet sind:

- Machbarkeitsstudie
- Arbeitskonzept
- Systemspezifikation
- endgültige Realisierung

Der grundlegende Fortschritt des Spiralmodells liegt in der Integration des zyklischen Charakters bei der Darstellung des Entwurfsprozesses. Jeder Durchlauf ermöglicht die direkte Einbeziehung des Entwicklers zur Durchführung

von Tests und Untersuchung von Lösungsalternativen. Auf diese Weise können Systemanforderungen iterativ konkretisiert werden. Hierbei ist zu erwähnen, dass zusätzliche Iterationen den Prototyp verbessern, jedoch schnell einen termingerechten Projektabschluss verhindern. Ein weiterer Nachteil liegt darin begründet, dass für jeden Zyklus alle Phasen durchlaufen werden müssen, was nicht immer machbar bzw. sinnvoll ist.

Das in Abbildung 2.5 dargestellte *V-Modell* [Forsberg u. a. 1996] beschreibt ausschließlich die technische Ebene der Systementwicklung. Den linken Teil der V-Form bilden Dekomposition und Definition, den rechten Integration und Verifikation, wobei die vertikale Achse den Detaillierungsgrad und die horizontale die fortschreitende Zeit darstellt. Das V-Modell erzwingt keine strikte zeitliche Abfolge, lediglich Aktivitäten und Ergebnisse werden definiert. Durch Dekomposition wird das System auf jeder Ebenen in mehrere Komponenten zerlegt, weswegen man auch von einem Top-Down-Entwurf sprechen kann. Dabei werden die folgenden Entwurfsschritte zur Definition der Systemspezifikation vertikal durchlaufen:

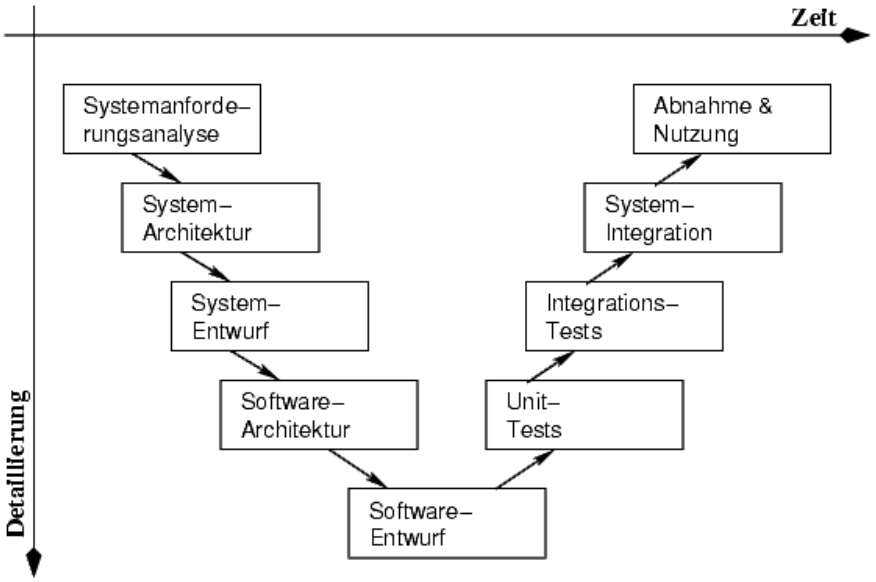

Abbildung 2.5: V-Modell

- Anforderungsanalyse, Erstellung eines Konzepts, Erstellung eines Validierungsplans

- Entwicklung einer Systemspezifikation bzw. -architektur inklusive Verifikationsplan
- Erweiterung der Systemspezifikation zur Entwurfsspezifikation einzelner Komponenten inklusive Verifikationsplan
- Umsetzung der Entwurfsspezifikationen in Softwarearchitektur inklusive Verifikationsplan

Die Basis des V stellt die Implementation der Komponenten anhand der abgeleiteten Softwarearchitektur dar. Im Anschluss an die Systemspezifikation wird die aufsteigende rechte Seite des V durchlaufen. Zuvor definierte Komponenten werden unter Einhaltung der Verifikationspläne zu einem Gesamtsystem integriert. Bei Nichterfüllen eines Tests erfolgt ein Absteigen in ein neues V mit aktualisierter Spezifikation. Ein Beispiel für den Einsatz des V-Modells stellt die Problem Frame Methode [Jackson 2001] dar, welche auf die Anforderungsanalyse sowie die Spezifikation einer Systemarchitektur fokussiert. Das V-Modell eignet sich hauptsächlich für Anwendungsfelder mit statischen Anforderungen [Wideman 2003], da dynamische Projekte eine permanente zeitaufwändige und damit kostenintensive Aktualisierung der Spezifikationsdokumentation verlangen.

Fasst man die charakteristischen Entwurfsschritte der genannten traditionellen Entwurfsansätze zusammen, lässt sich ein gemeinsames Schema ableiten. Nach der Konzeptentwicklung und der Erstellung einer geschriebenen Spezifikation, wird der Entwurf in Subsysteme zerlegt, um anschließend jedes Subsystem isoliert zu betrachten. Konflikte, wie beispielsweise durch gemeinsam genutzte Ressourcen oder dynamische Kopplungseffekte, werden zunächst ignoriert. Nachdem jedes Subsystem entworfen wurde, werden diese wieder zusammengeführt und etwaig auftretende Konflikte durch systematische Vergleiche aufgelöst. Letztlich wird das entwickelte Produkt freigegeben. Abbildung 2.6 fasst das erläuterte Schema bezüglich Produktunsicherheit und Entwicklungskosten zusammen. Man erkennt, dass die Produktunsicherheit mit dem Fortschreiten des Entwurfsprozesses abnimmt. Gleichzeitig steigen die Entwicklungskosten bis zur Freigabe des Produktes immens an. Dabei treten die Hauptkosten während der Integration auf, da die Konflikte zwischen den Subsystemen erst jetzt behandelt werden.

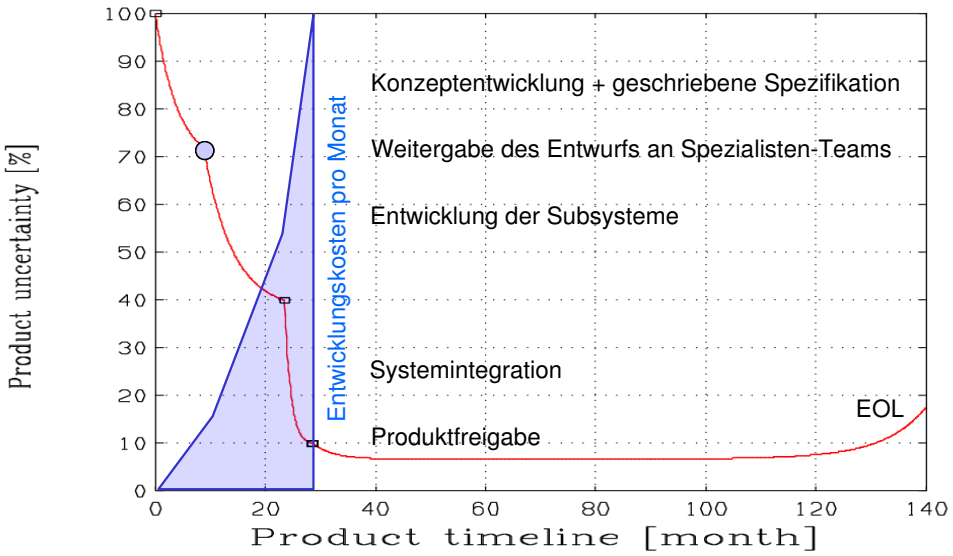

Abbildung 2.6: Produktunsicherheit und Entwicklungskosten beim traditionellen Entwurf [Salzwedel u. a. 2009]

2.3.2 Mission Level Design

Eine der jüngsten Entwicklungen im Bereich des Systementwurfs stellt das *Mission Level Design* [Schorcht 2000] dar. Ausgangspunkt bildet die Analyse der operationalen Anforderungen an das System, um eine Spezifikation inklusive typischer Einsatzszenarien des Systems, alias Missionen, abzuleiten. Die Spezifikation dient als Grundlage zur Entwicklung eines ausführbaren Gesamtsystemmodells, welches Funktion, Architektur und Umgebung bzw. Mission beinhaltet. Dieses wird gegen die Gesamtsystemanforderungen validiert und optimiert, wodurch die Anforderungen an die Subsysteme konkretisiert werden können. Daraufhin wird die ausführbare Gesamtsystemspezifikation an mehrere Teams zur Entwicklung der Subsysteme weitergegeben. Da die Gesamtsystemspezifikation alle Subsysteme auf abstraktem Niveau enthält, können Integrationsprobleme durch Kopplungseffekte frühzeitig erkannt und gelöst werden. Nach der Entwicklung werden die Subsysteme wieder zu einem Gesamtsystem integriert werden. Abbildung 2.7 zeigt die Entwurfsebenen des Mission Level Designs im Überblick.

Um eine frühzeitige Entscheidungsfindung zu ermöglichen, liegt der Ent-

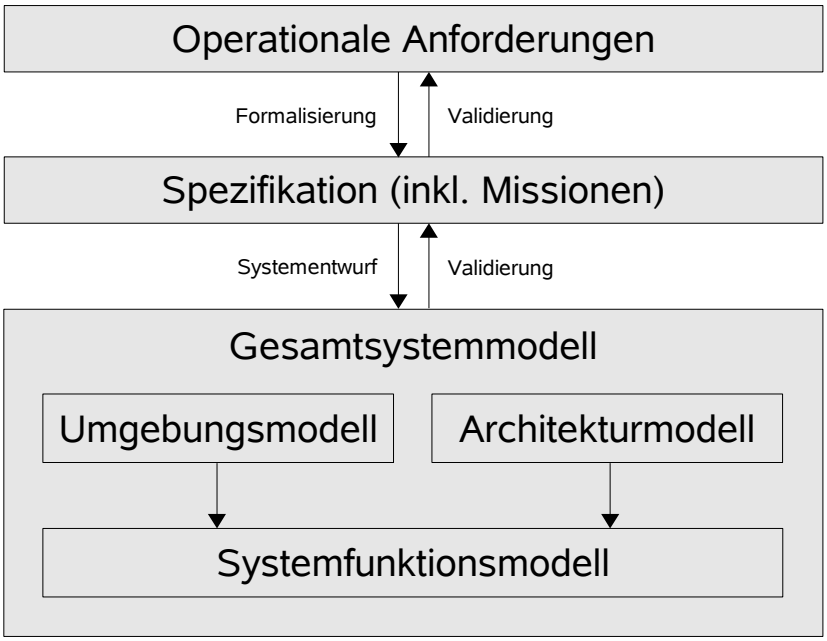

Abbildung 2.7: Mission Level Design [Schorcht 2000]

wurfsschwerpunkt des Mission Level Designs in der Spezifikationsphase. Dies bedeutet, dass nicht die Implementierung, sondern die Spezifikation und strukturierte Planung des Systems im Vordergrund steht. Dabei wird als Mittel zur Analyse, Verifikation und Validierung die Simulation eingesetzt [Kalavade u. Lee 1996]. Grund hierfür ist die Tatsache, dass die frühen Entwurfsphasen durch einen Mangel an präzisen Informationen geprägt sind und formale Methoden grundsätzlich keine semantische Aussage über die Erfüllung der Systemfunktion erlauben [Schorcht 2000]. Voraussetzung zur Simulation des Gesamtsystemmodells ist die Ableitung und Validierung von Einsatzszenarien, alias Missionen. Diese definieren Eingabesequenzen, sowie zu erwartende Systemantworten. Hierbei kommt der Missionsauswahl eine erhebliche Bedeutung bezüglich der Testergebnisse zu. Nur wenn typische Varianten aus dem gesamten Einsatzspektrum des Systems selektiert werden, kann auch eine sinnvolle Validierung erfolgen. In diesem Zusammenhang wird die Missionsanzahl einerseits durch die Simulationsdauer und andererseits durch die verfügbaren Auswertungsmöglichkeiten limitiert.

Das Mission Level Design ermöglicht bereits in den frühen Entwurfsphasen

die Auffindung und Lösung von Integrationsproblemen auf Grundlage einer ausführbaren Gesamtsystemspezifikation. Hierdurch kann das Entwicklungsrisiko gesenkt werden. Abbildung 2.8 verdeutlicht diesen Zusammenhang in Bezug auf Produktunsicherheit und Entwicklungskosten genauer. Es ist zu erkennen, dass durch den Einsatz einer ausführbaren Gesamtsystemspezifikation die Produktunsicherheit frühzeitig gesenkt werden kann, da die Subsystem-Entwickler beim Eigenentwurf auf ein validiertes Gesamtsystem zurückgreifen können. Gleichzeitig wird hierdurch der Entwurfsprozess beschleunigt und eine frühere Produktfreigabe erreicht.

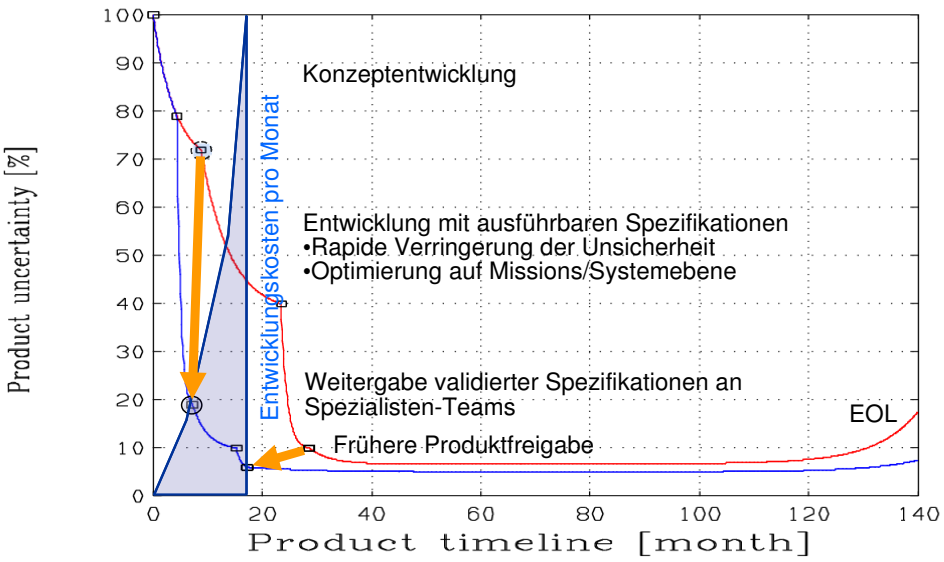

Abbildung 2.8: Produktunsicherheit und Entwicklungskosten beim Mission Level Design [Salzwedel u. a. 2009]

2.4 Bewertung

Im weiteren Verlauf wird der traditionelle Entwurf und das Mission Level Design hinsichtlich der Problemstellung aus Abschnitt 1.2 und der Entwurfsansatzanforderungen aus Abschnitt 2.2 analysiert und bewertet. Anschließend werden Ansatzpunkte zur Erhöhung der Spezifikationsqualität und -geschwindigkeit für das Mission Level Design abgeleitet.

Beim Entwurfsansatz Mission Level Design werden Systeme modellbasiert in Form von ausführbaren Gesamtsystemspezifikationen beschrieben. Nach der Anforderungsanalyse wird ein Gesamtsystemmodell, bestehend aus einer Funktions-, Architektur- und Umgebungs- bzw. Missionsbeschreibung unter Verwendung unterschiedlicher Ausführungsmodelle entworfen. Das Gesamtsystemmodell dient als Grundlage zur simulativen Analyse und Validierung des Systems am virtuellen Prototypen, wodurch die Entscheidungsfindung in die frühen Entwurfsphasen verlagert werden kann. Beim traditionellen Entwurf ist dies nicht der Fall. Dort werden auf Gesamtsystemebene lediglich geschriebene, nicht validierbare Spezifikationen eingesetzt, welche anschließend als Ausgangspunkt für den Subsystem-Entwurf dienen. Folglich liegt der Entwurfsschwerpunkt in der Entwurfs- und nicht in der Spezifikationsphase. Dabei kommen für den Entwurf der Subsysteme größtenteils ausführbare Modelle zum Einsatz.

Ein weiteres Manko des traditionellen Entwurfs ist die Zerlegung komplexer Entwurfsprobleme in Teilprobleme unter Nichtbeachtung der Kopplungseffekte. Als Beispiel hierfür kann die Problem Frame Methode [Jackson 2001] angeführt werden. Dort werden Probleme zerlegt und die Teilprobleme jeweils isoliert betrachtet. Konflikte werden erst am Ende des Entwurfs während der Integration berücksichtigt. Im Gegensatz dazu werden beim Mission Level Design die Kopplungseffekte zu Beginn des Entwurfs auf Gesamtsystemebene untersucht, um die Anforderungen an die Subsysteme zu konkretisieren, bevor diese entworfen werden. Hierdurch kann die Produktunsicherheit frühzeitig verringert werden (siehe Abbildung 2.6 und 2.8).

In diesem Zusammenhang stellt die Auffindung einer optimalen Architektur auf Gesamtsystemebene bezüglich der Subsysteme eine besondere Herausforderung dar [Salzwedel u. a. 2008]. Voraussetzung ist die Möglichkeit zur iterativen Zuordnung von Funktion auf Architektur und umgekehrt, was wiederum eine separate Funktions- und Architekturentwicklung mit anschließender Bindung und Leistungbewertung voraussetzt. Der traditionelle Entwurf unterstützt eines solches Konzept aufgrund des Fehlens einer ausführbaren Gesamtsystemspezifikation nicht. Beim Mission Level Design ist eine Gesamtsystemoptimierung prinzipiell möglich. Bedingung ist die Einführung einer standardisierten Architekturbeschreibung zur formalen Trennung zwischen den Abstraktionsebenen für Funktion, Architektur und Umgebung, sowie zum Mapping zwischen Funktion und Architektur. Grundlagen hierfür wurden bereits in [Paluch 2004] und [Zens 2003] untersucht, jedoch wurde keine allgemeingültige Lösung abgeleitet. Dabei muss die Architekturbeschreibung analog zur Funktions- und Umgebungsbeschreibung ein flexibles Abstrakti-

onsniveau unterstützen, um unterschiedliche Entwurfsziele zu ermöglichen. Zudem ist eine zusätzliche Entwurfsebene zur Analyse unterschiedlicher Kombinationen von Funktion und Architektur einzuführen.

Um die Optimierung auf Gesamtsystemebene zu beschleunigen, muss diese automatisiert werden. Das Mission Level Design unterstützt zwar einen iterativen Entwurfsvorgang anhand von Modelländerungen und Modellverfeinerungen, jedoch lediglich manuell. Eine automatisierte Entwurfsraumanalyse zur Optimierung ist nicht durchführbar. Durch Einsatz eines ausführbaren Entwurfsprozesses können solche Vorgänge automatisiert werden. Voraussetzung ist die Schaffung einer Möglichkeit zur iterativen Generierung von Modellen während der Simulation, sowie einer vereinheitlichten Bewertungsmöglichkeit der System-Performance-Analyse.

Auch die Durchführung einer Synthese auf Basis des Gesamtsystems und der dort festgelegten Subsystem-Architektur gestaltet sich aufgrund der nicht vorhandenen standardisierten Architekturbeschreibung als schwierig. Zwar ist eine Synthese grundsätzlich möglich, wie in [Rath u. Salzwedel 2004] für interagierende Zustandsmaschinen gezeigt wurde, jedoch fehlt ein gemeinsamer Ausgangspunkt zur Festlegung der Architektur und der jeweiligen Architektureigenschaften. Dies bedeutet, dass weder verschiedene Architekturen festgelegt, noch zwischen zu synthetisierenden und nicht zu synthetisierenden Modellkomponenten unterschieden werden kann. Hinzu kommt das Fehlen von Transformationsvorschriften, ohne die ein automatisierter Übergang zur Implementationsebene nicht möglich ist [Baumann u. a. 2007a].

Die Ergebnisse der Bewertung werden in Tabelle 2.1 zusammengefasst. Es ist zu erkennen, dass das Mission Level Design im Gegensatz zum traditionellen Entwurf bereits eine Vielzahl der Entwurfsansatzanforderungen erfüllt. Zudem ist zu erkennen, dass mehrere Ansatzpunkte zur Weiterentwicklung des Mission Level Design existieren:

- Optimierung auf Gesamtsystemebene
- Ausführbarer Entwurfsprozess
- Dynamische Modellgenerierung
- Synthese auf Basis des Gesamtsystems

Im weiteren Verlauf der Arbeit ist abzuklären, wie die Entwurfsansatzanforderungen durch das Mission Level Design erfüllt werden können. Hierzu muss zunächst der aktuelle Entwurfsprozess inklusive Entwurfsebenen analysiert

Entwurfsansatzanforderung	**Traditioneller Entwurf**	**Mission Level Design**
Anforderungsanalyse	x	x
Entwurfsschwerpunkt Spezifikationsphase	–	x
Entwurf auf Gesamtsystemebene	–	x
Modellbasierter Entwurf	x	x
Ausführbare Spezifikation	x	x
Multidomänen-Entwurf	–	x
Variables Abstraktionsniveau	x	x
Integrierter Entwurf	–	x
Optimierung auf Gesamtsystemebene	–	–
Ausführbarer Entwurfsprozess	–	–
Dynamische Modellgenerierung	–	–
Synthese auf Basis des Gesamtsystems	–	–

Tabelle 2.1: Erfüllung der Entwurfsansatzanforderungen

und angepasst werden. Anschließend ist abzuklären wie die vorgenommenen Erweiterungen implementiert werden können.

Kapitel 3

Automatisierung des Mission Level Designs

3.1 Grundgedanke

Nachdem in Abschnitt 2.4 eine Analyse und Bewertung des Mission Level Designs durchgeführt wurde, wird im weiteren Verlauf der Entwurfsansatz bezüglich der nicht erfüllten Entwurfsansatzanforderungen erweitert. Dabei wird die Zielstellung verfolgt, die Spezifikationsqualität und -geschwindigkeit durch Erhöhung des Automatisierungsgrads weiter zu steigern.

Um eine Optimierung auf Gesamtsystemebene zu ermöglichen, muss das Mission Level Design eine Iteration über unterschiedliche Entwurfsalternativen bzw. virtuelle Prototypen unterstützen. Hierzu muss einerseits eine Entwurfsebene zur Repräsentation der aktuellen Entwurfsalternative, sprich einer Zuordnung von Funktion auf Architektur oder umgekehrt eingeführt werden. Andererseits muss ein Prozess zur Erzeugung, Validierung und vereinheitlichten Performance-Analyse unterschiedlicher Entwürfe in einem Missionsumfeld definiert werden. Dieser bildet gleichzeitig die Grundlage zur Einführung eines ausführbaren Entwurfsprozesses. In diesem Zusammenhang ermöglicht die Performance-Analyse eine frühzeitige Abschätzung der Leistungsfähigkeit von Systemarchitekturen am virtuellen Prototypen.

Des weiteren muss auf Grundlage der auf Gesamtsystemebene beschriebenen Architektur des virtuellen Prototyps eine plattformspezifische Synthese unterstützt werden. Nur so kann der virtuelle Prototyp in einen realen Prototyp überführt werden. Hierzu muss das Mission Level Design um eine Implemen-

tationsebene, die mit der Ebene des aktuellen Entwurfs in Beziehung steht, erweitert werden.

Im Rahmen der folgenden Abschnitte werden zunächst die Entwurfsebenen und der Entwurfsprozess definiert. Anschließend werden Untersuchungen zur Ausführung des Entwurfsprozesses durchgeführt. Anzumerken ist, dass der Entwurfsprozess mit seiner iterativen Entwurfsraumanalyse nicht nur für technische Systeme eingesetzt werden kann. Beispielsweise eignet er sich auch für ökonomische Systeme [Schneider u. a. 2005].

3.2 Entwurfsebenen

Beim Mission Level Design werden mehrere interagierende Entwurfsebenen unterschieden. Grundlage bildet eine ausführbare Gesamtsystemspezifikation, welche unmittelbar aus den operationalen Systemanforderungen abgeleitet wird. Diese beinhaltet zum einen die Funktion und Architektur des zu entwerfenden Systems und zum anderen dessen Umgebung inklusive Einsatzszenarien, alias Missionen. In diesem Zusammenhang wird eine konkrete Zuordnung bzw. Partitionierung von Funktion auf Architektur als virtueller Prototyp bezeichnet. Um über verschiedene virtuelle Prototypen zur Auffindung eines möglichen oder optimalen Entwurfs zu iterieren, muss ein Prozess zur Erzeugung, Validierung und Bewertung virtueller Prototypen eingeführt werden. Zudem muss eine Entwurfsebene zur Repräsentation des aktuellen virtuellen Prototyps definiert werden, welche im weiteren Verlauf als *Electronic System Level* (ESL) bezeichnet wird. Auf Basis des aktuellen virtuellen Prototyps kann im Anschluss eine Transformation zur Implementationsebene durchgeführt werden. Die Entwurfsebenen des erweiterten Mission Level Design werden in Abbildung 3.1 dargestellt und im folgenden erläutert:

- **Operationale Anforderungen:**
  Hier werden die möglichen bzw. relevanten Betriebsbedingungen eines Systems erfasst. Im weiteren Verlauf dienen diese als Ausgangspunkt zur Ableitung von Missionen sowie zur Konkretisierung weiterer funktionaler und nichtfunktionaler Anforderungen.

- **Ausführbare Gesamtsystemspezifikation:**
  Auf Ebene der Gesamtsystemspezifikation werden Funktion und Architektur des Systems inklusive der Subsysteme, sowie dessen Umgebung und Missionen in ausführbarer Form erfasst.

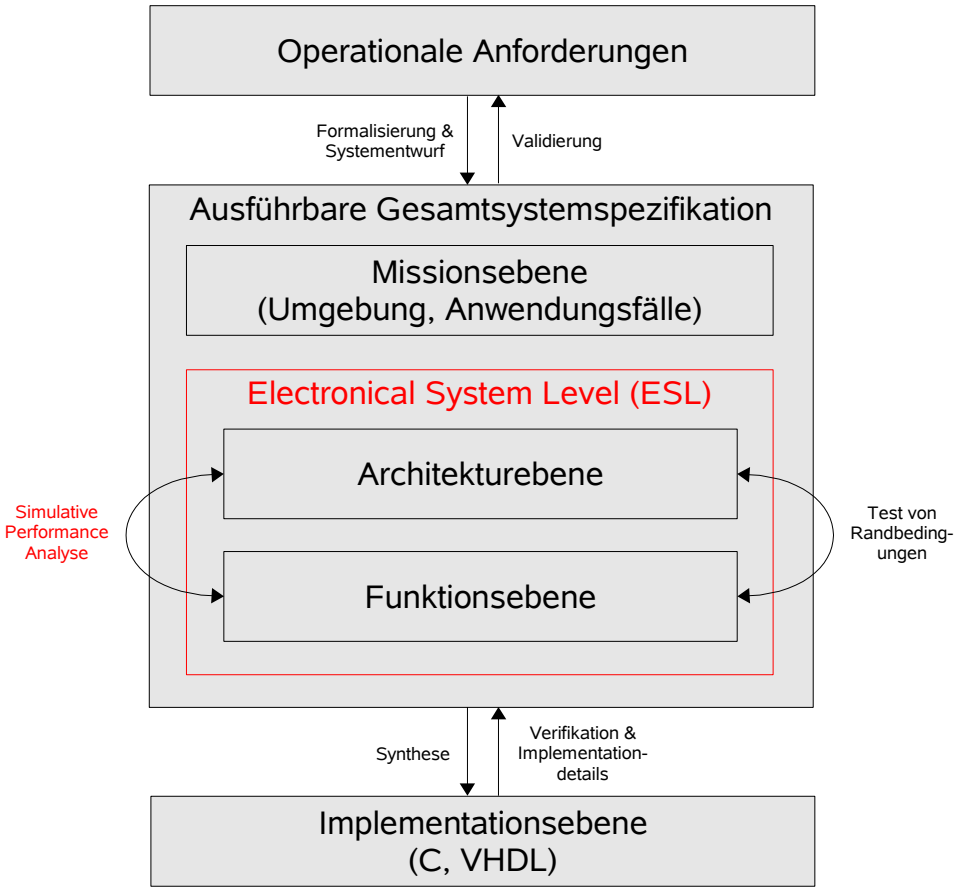

Abbildung 3.1: Erweitertes Mission Level Design

- **Missionsebene:**
  Auf dieser Ebene wird das Umgebungsmodell mit seinen Aktoren und Anwendungsfällen bzw. Missionen beschrieben.

- **Funktionsebene:**
  Die Funktionsebene beschreibt das eigentliche zu entwerfende System ausgehend von den funktionalen Anforderungen in idealisierter Form, das heißt ohne Performance-Eigenschaften.

- **Architekturebene:**
  Ausgehend von den nichtfunktionalen Anforderungen werden auf Architekturebene die Ausführungs- und Kommunikationsstrukturen des

Systems funktionsunabhängig beschrieben. Da die Ebene als Grundlage zur Bewertung der System-Performance dient, wird sie auch als Performance-Ebene bezeichnet.

- **Electronic System Level:**
  Die neu eingeführte ESL-Ebene verbindet bereits in den frühen Entwurfsphasen Funktion und Architektur zu virtuellen Prototypen, wodurch eine iterative Validierung und Bewertung ermöglicht wird.

- **Implementationsebene:**
  Unter der Implementationsebene versteht man die konkrete Realisierung des virtuellen Prototyps in Form von plattformspezifischen Hard- und Softwarebeschreibungen.

3.3 Entwurfsprozess

Im weiteren Verlauf wird der Entwurfsprozess des erweiterten Mission Level Designs definiert. Hierzu werden die einzelnen Entwurfsschritte benannt und erläutert. Abbildung 3.2 stellt den Entwurfsprozess in Form eines Flussdiagramms dar. Anzumerken ist, dass der Entwurfsprozess auch Rücksprünge zu unterschiedlichen Einstiegspunkten unterstützt, was durch Signale veranschaulicht wird.

Ausgangspunkt des Entwurfs bildet eine Anforderungsanalyse zur Ermittlung funktionaler und nichtfunktionaler Anforderungen. Dabei dienen die operationalen Anforderungen, das heißt die Betriebsbedingungen des zu entwerfenden Systems, als Grundlage zur Ableitung von Anwendungsfällen bzw. Missionen. Diese bilden zusammen mit der Funktion und gegebenenfalls feststehenden Architektur ein ausführbares, validierbares Gesamtsystemmodell. Um bereits in den frühen Entwurfsphasen eine optimale Architektur für das Gesamtsystem zu ermitteln, wird im nächsten Schritt eine Iteration über Architekturalternativen durchgeführt. Voraussetzung ist die Erzeugung von Gesamtsystemmodellen mit unterschiedlichen Architekturen. Hierzu muss zunächst eine Architektur festgelegt (Allokation) und anschließend die Funktion auf diese verteilt werden (Bindung), was im Allgemeinen auch als Partitionierung bezeichnet wird. Wie später noch gezeigt wird, können solche Modelle auf Grundlage einer standardisierten Architekturbeschreibung, sowie einer steuernden Annotierungssprache automatisiert erzeugt werden. Im Anschluss wird das erzeugte Gesamtsystemmodell zur Validierung und Bewertung ausgeführt. Dies bedeutet, dass die Performance-Eigenschaften des aktuellen vir-

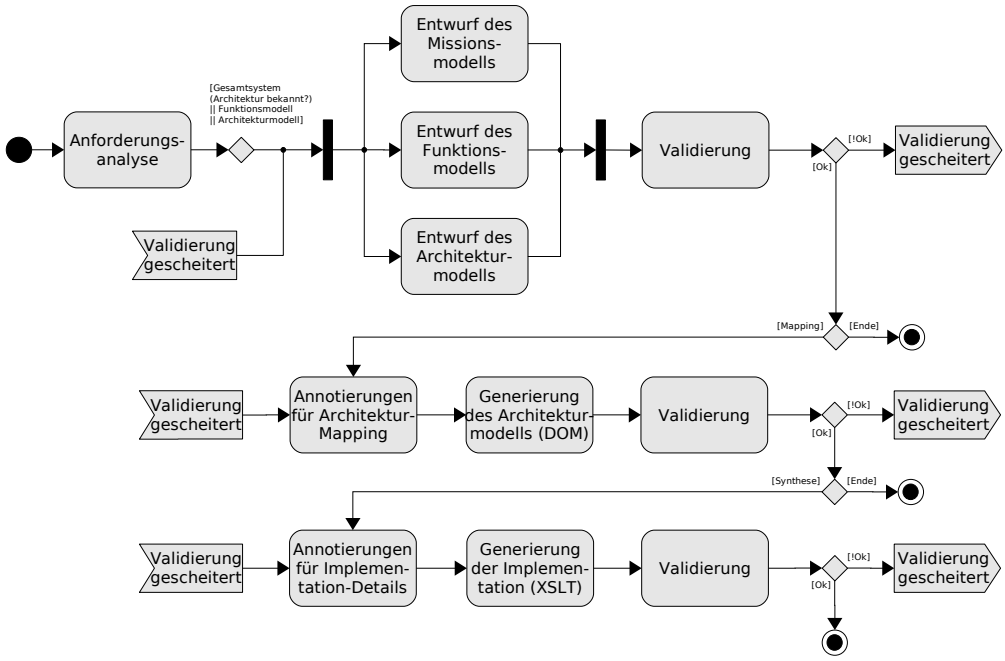

Abbildung 3.2: Flussdiagramm des Entwurfsprozesses

tuellen Prototyps, wie beispielsweise Zeiten, Fehlerraten, Durchsatzraten und Auslastung von Ressourcen, analysiert werden. Wird dabei festgestellt, dass der virtuelle Prototyp invalid oder schlechter als bereits bewertete ist, wird dieser verworfen und ein neues Gesamtsystemmodell erzeugt. Ein solch iteratives Durchlaufen des Entwurfsraums wird auch als Entwurfsraumanalyse bezeichnet.

Sobald die Optimierung oder auch lediglich Partitionierung auf Gesamtsystemebene abgeschlossen ist, wird mit dem Entwurf der Subsysteme fortgefahren. Hierzu werden die Subsystem-Spezifikationen unter Verwendung der Gesamtsystemspezifikation konkretisiert und auf Teams von Spezialisten verteilt. Daraufhin werden diese entworfen, validiert und wieder zu einem Gesamtsystem integriert. Da hierbei die Subsysteme weiter verfeinert und zergliedert werden, wird das Mission Level Design auch als *Top-Down-Entwurf* klassifiziert.

Anschließend kann der Entwurf auf Grundlage der im Gesamtsystemmodell bzw. virtuellen Prototyp festgelegten Architektur inklusive zugeordneter Funktionen in mehrere plattformspezifische Implementationen überführt wer-

den. Da der virtuelle Prototyp auf abstrakter Ebene entworfen wurde und somit Details über die Implementation fehlen, muss das Modell zuvor verfeinert werden. Im Verlauf der Arbeit wird gezeigt, dass dies durch Zuordnung plattformspezifischer Annotierungen an die standardisiert entworfenen Architekturkomponenten erreicht werden kann. Wurden diese ergänzt, kann der eigentliche Syntheseprozess gestartet werden. Dabei werden die mittels der Extensible Markup Language (XML) gespeicherten Modelle unter Verwendung der Extensible Stylesheet Language for Transformation (XSLT) transformiert [Baumann u. a. 2007a]. Dies bedeutet, dass die Architekturkomponenten inklusive der zugeordneten Funktionen zu C-Code oder in VHDL repräsentierte Intellectual Property Cores (IP-Cores) überführt werden. Dabei können entweder neue IP-Cores synthetisiert oder bereits existierende eingebunden werden. Schlussendlich wird die synthetisierte Implementation getestet und gegen die gestellten Anforderungen validiert. Ist die Validierung nicht erfolgreich, muss eine weitere Iteration zur Verfeinerung des abstrakten Modells durch Annotierungen oder eine direkte Abänderung an der Implementation durchgeführt werden.

3.4 Ausführbarer Entwurfsprozess

Der in Abschnitt 3.3 vorgestellte Entwurfsprozess zeichnet sich durch mehrere sequenzielle und iterative Entwurfsschritte aus. Zwar sind einzelne Entwurfsschritte automatisiert durchführbar, wie beispielsweise die Erzeugung von Gesamtsystemmodellen mit unterschiedlichen Architekturen und die plattformspezifische Synthese, jedoch nicht der Entwurfsprozess selbst. Konkret bedeutet dies, dass die Übergänge zwischen den iterativen Entwurfsschritten manuell erfolgen, woraus ein erhöhter Zeitbedarf resultiert. Zur Beschleunigung der Übergänge muss der Entwurfsprozess zur Ausführung gebracht werden. Nur so können Entwurfsraumanalysen zur Ermittlung möglicher oder optimaler Entwurfsalternativen, sowie Worst-Case-Analysen zur Bemessung der Systemgrenzen automatisiert durchgeführt werden. Zudem kann der Entwurfsprozess simulativ validiert werden [Salzwedel 2007]. Anzumerken ist, dass der ausführbare Entwurfsprozess an die jeweilige Problemstellung angepasst werden muss, weswegen gegebenenfalls weitere Iterationsschleifen hinzuzufügen sind.

Zur Ausführung des Entwurfsprozess muss dieser modellbasiert erfasst, sowie eine Ausführungsdomäne festgelegt werden. Hierfür eignen sich beispielsweise Aktivitätsdiagramme und Blockflussdiagramme, da dort Sequenzen und

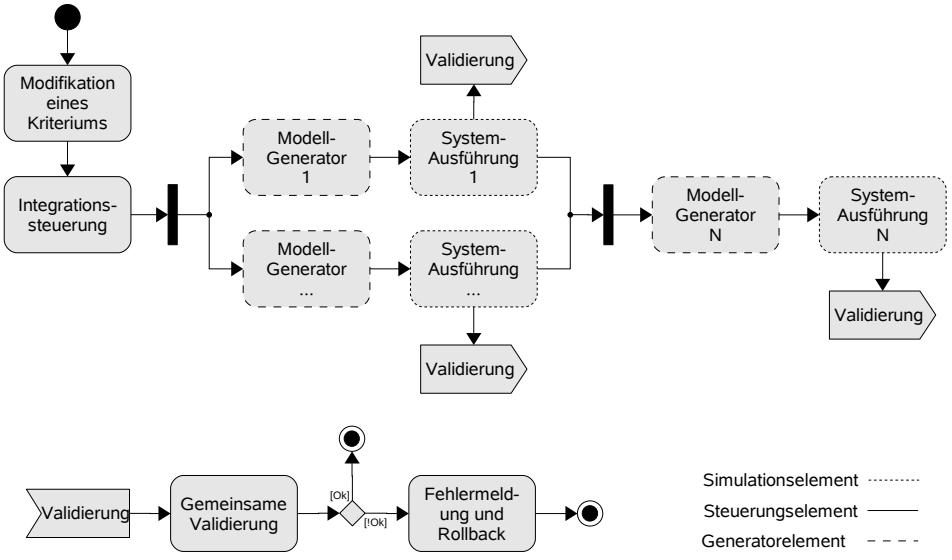

Abbildung 3.3: Prozessgesteuerte Modifikation des Entwurfsraums

Iterationen abgebildet werden können. Um die Semantik der unterschiedlichen Entwurfsschritte beschreiben zu können, sind folgende grundlegende Modellelemente erforderlich:

- *Simulationselemente* definieren die Ausführung der Gesamtsystemspezifikation oder von Subsystemen aus einer bestimmten anforderungsorientierten Perspektive. Hierbei sind auch abstrakte Systeme möglich, deren Simulationsergebnisse die Generierung konkretisierter Systeme beeinflusst.

- *Steuerungselemente* werten Regeln und Zielfunktionen zur Steuerung des Prozesses aus, um gegebenenfalls weitere Iterationen anzustoßen. Ein möglicher Auslöser ist die Abänderung einer Systemeigenschaft, welche die Überprüfung oder Abänderungen weiterer Systemeigenschaften erfordert. Ursache hierfür sind Abhängigkeiten zwischen den an das System und dessen Subsysteme gestellten Anforderungen. Ein weiterer Auslöser können iterativ zu durchlaufende Optimierungszyklen sein, welche anhand von Zielfunktionen gesteuert werden.

- *Generatorelemente* erstellen Modelle mit verschiedener Strukturen und Verhalten, sowie Parametrisierungen für existierende Modelle. Die Ge-

nerierung findet während der Prozessausführung statt, um eine sofortige Integration und Ausführung zu ermöglichen. Grundlage bilden Informationen aus anderen Modellen sowie Nutzereingaben.

Abbildung 3.3 stellt die Anwendung der Elemente zur prozessgesteuerten Modifikation des Entwurfsraums, sowie die Umsetzung in Form von Aktivitäten dar. Als Ausgangspunkt dient die Modifikation eines anforderungsspezifischen Entwurfskriteriums durch den Entwickler, woraufhin mehrere Aktualisierungen und Tests bezüglich abhängiger Entwurfskriterien durchzuführen sind. Diese können sequenziell oder insofern keine Abhängigkeit existieren auch parallel durchgeführt werden. Ziel ist es, die Modifikation hinsichtlich der Erhaltung eines validen Entwurfszustands zu beurteilen, um zu entscheiden, ob diese akzeptiert werden können oder zu verwerfen sind. Ein vergleichbarer iterativer Ansatz wird bei der Optimierung von Entwurfskriterien verwendet. Abbildung 3.4 stellt den Prozessfluss dar, wobei auch hier die zuvor definierten Aktivitäten zu finden sind. Jede Entwurfsoptimierung besteht prinzipiell aus einer Determinierungs-, Simulations- und Bewertungsphase. Vorteil der Trennung ist die leichte Austauschbarkeit dieser Komponenten, genauer gesagt des zu optimierenden Problems bzw. Entwurfs, der Optimierungsheuristik [Feldmann 1999], sowie der Gütefunktion. Der Optimierungsprozess ist beendet, sobald eine zuvor festgelegte Iterationsanzahl oder Entwurfsgüte erreicht wird.

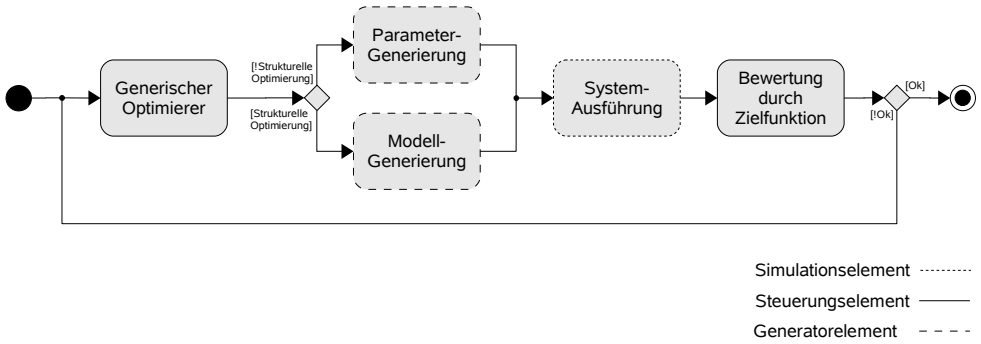

Abbildung 3.4: Prozessgesteuerte Optimierung des Entwurfsraums

Kapitel 4

MLDesigner-Anwendungsumgebung

4.1 Grundgedanke

In den folgenden Abschnitten werden Voraussetzungen und Möglichkeiten zur Realisierung der in Kapitel 3 eingeführten Erweiterungen des Mission Level Designs diskutiert. Ziel ist es ein geeignetes Konzept abzuleiten. Hierzu sind die Eigenschaften und Anforderungen der erläuterten Entwurfsebenen sowie des Entwurfsprozesses zu analysieren. Grundsätzlich lassen sich zwei Rahmenbedingungen im Vorab festlegen:

- Es sollen Erweiterungen zur Automatisierung des Mission Level Designs umgesetzt werden, weswegen sich der Einsatz des Systementwurfswerkzeugs MLDesigner anbietet. MLDesigner unterstützt bereits den Entwurfsansatz in seiner aktuellen Form und erfüllt somit implizit einen Großteil der Entwurfsansatzanforderungen.
- Es soll eine standardisierte Architekturbeschreibung für verteilte Systeme entwickelt werden, da diese die Grundlage zur iterativen Architekturoptimierung inklusive Modellgenerierung, zur vereinheitlichten Performance-Analyse, sowie zur Synthese darstellt.

Basierend auf den Rahmenbedingungen lässt sich die Idee zur Entwicklung einer *MLDesigner-Anwendungsumgebung zur Beschreibung und Analyse verteilter Systemarchitekturen* folgern, wobei der Schwerpunkt auf der Durchführung frühzeitiger Performance-Analysen liegt. In diesem Zusammenhang

wird unter einer Anwendungsumgebung eine systematische Sammlung von Anwendungskomponenten inklusive Verbindungsmechanismen im Sinne eines Gesamtkonzepts verstanden.

Im weiteren Verlauf wird das Systementwurfswerkzeug MLDesigner eingeführt und hinsichtlich des Einsatzes für die geplanten Erweiterungen untersucht. Insbesondere wird auf die verwendeten Modellelemente und Ausführungsdomänen eingegangen. Anschließend werden grundlegende Eigenschaften verteilter Systeme analysiert. Dabei stehen hauptsächlich Verhalten und Struktur von Ausführungs- sowie Kommunikationskomponenten im Vordergrund, wie beispielsweise interagierende Protokolle, Übertragungsmedien und Netzwerktopologien.

4.2 Systementwurfswerkzeug MLDesigner

4.2.1 Überblick

MLDesigner [MLDe 2008] ist eine integrierte Plattform zur Modellierung und Analyse von Systemen und hat seine Wurzeln im freien Simulationswerkzeug Ptolemy-Classic [Univ 1998]. Es ermöglicht die Anwendung des in Abschnitt 2.3.2 erläuterten Entwurfsansatzes *Mission Level Design*. Hierbei kommen verschiedene Modelle zur Beschreibung von Systemstruktur und -verhalten in Form von ausführbaren Spezifikationen zum Einsatz. Ziel ist es, ausgehend von dieser Spezifikation, Rückschlüsse auf die Realität zu ziehen und gegebenenfalls Code zu synthetisieren. Abbildung 4.1 zeigt die grafische Oberfläche von MLDesigner, welche aus einem Frame-Set verschiedener Editoren besteht. Zu sehen sind der Multi-Dokument-Editor für den graphischen Entwurf von Modellen (zentral), eine Baumstruktur zur Modellauswahl (links oben), ein Parametereditor (links unten), der Datenstruktureditor (rechts) und die Kommandozeile (unten).

Zur Modellierung werden im wesentlichen hierarchisch gekoppelte Blockflussdiagramme verwendet, welche größtenteils den Kompositionsstrukturdiagrammen der UML 2.0 entsprechen (siehe Abbildung 2.1). Blockflussdiagramme stellen Systemkomponenten in Form von interagierenden Funktionseinheiten bzw. Blöcken dar [Grüner 2007]. Jeder Block verarbeitet seine Eingaben unter Berücksichtigung des internen Zustands und der Parametrisierung zu Ausgaben, wobei der Datenaustausch untereinander über definierte Schnittstellen, den sogenannten Ports, stattfindet. Dabei können Blöcke durch unterschiedliche Modelle beschrieben und auch mehrfach instanziiert werden. Je-

des Modell kann mehrere typisierte Parameter definieren, welche bei Instanziierung neu gesetzt oder über verschiedene Hierarchieebenen hinweg verlinkt werden können. Die Datenhaltung der Modelle beruht auf einem XML konformen Format [Hauguth 2000].

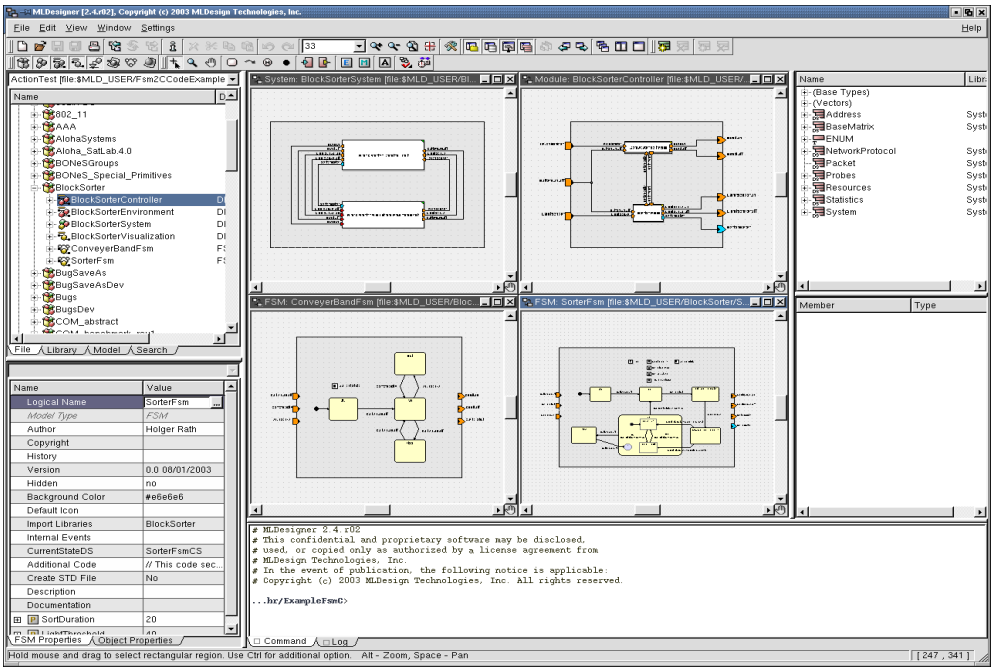

Abbildung 4.1: Benutzeröberfläche des MLDesigner

4.2.2 Modelle

Allgemein beschreibt ein Modell in MLDesigner die Struktur und das Verhalten eines Systems oder Teilsystems der realen Welt. Abbildung 4.2 zeigt die hierarchische Kombination solcher Modelle zur Dekomposition des Gesamtsystems in Komponenten bzw. Subsysteme. Im folgenden werden die grundlegenden Modelltypen erläutert:

- Ein *Primitive* ist ein Block, welcher sich aus keinen weiteren Blöcken zusammensetzt. Die Spezifikation erfolgt in MLDesigner im Allgemeinen in der Ptolemy Language (PTLang), welche eine relativ geringe

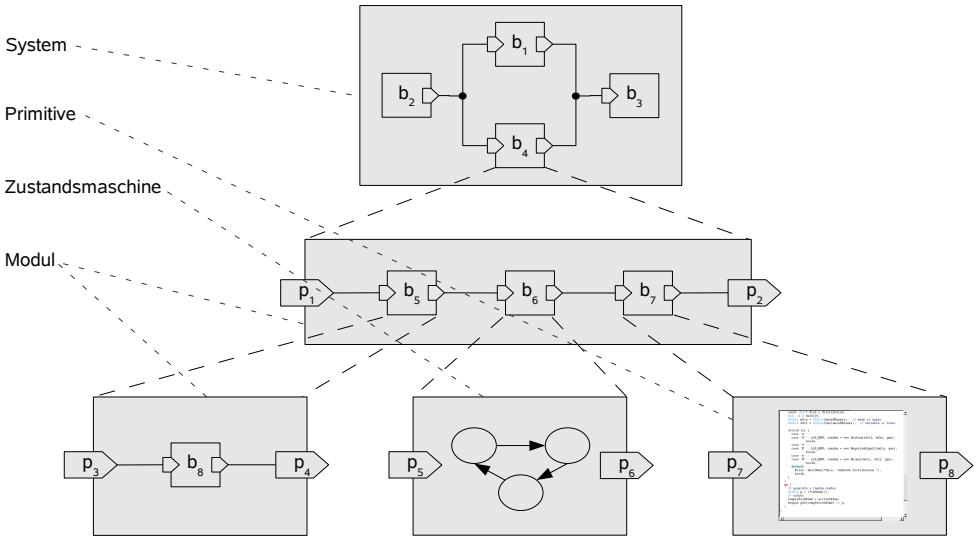

Abbildung 4.2: Hierarchische Modellanordnung in MLDesigner

Abstraktionsstufe einer C++ Klasse zur Beschreibung von Struktur und Verhalten darstellt. Hierbei wird die Strukturbeschreibung grafisch modelliert und automatisch nach PTLang übersetzt. Ein spezieller Primitive-Typ ermöglicht alternativ zu C++ Codefragementen und PTLang die Verhaltensbeschreibung durch Finite State Machines (FSM) bzw. Statecharts. Primitives kommunizieren mit der Außenwelt über typisierte Ports.

- Als *Modul* bezeichnet man das Modell eines Teilsystems, welches aus Blöcken zusammengesetzt ist. Blöcke sind hierbei als Instanzen bzw. Ausprägungen anderer Modelle zu verstehen, analog zum Begriff Instanz aus der objektorientierten Softwareentwicklung. Besitzt eine Instanz ein Modul als Modell, ergibt sich eine Baumstruktur, in welcher Modellbeziehungen auch als hierarchische Einbettung bezeichnet werden. Module kommunizieren genauso wie Primitives über typisierte Ports mit der Umgebung. Primitive und Modul präsentieren sich demnach eingebettet äquivalent.

- Ein *System* ist konzeptionell ein spezielles Modul ohne Ports. Es kann nur in eine Library eingebettet werden, also nicht Teil eines weiteren Systems sein. In MLDesigner sind nur Systeme ausführbar. Hierzu be-

sitzt es spezielle Parameter, wie etwa die Dauer einer Simulation.

- Eine *Library* gruppiert, verwaltet und versioniert Modellelemente. MLDesigner liefert umfangreiche Libraries bzw. Bibliotheken mit vordefinierten und validierten Primitives, so dass die Konzentration eines Systemdesigners im Bereich der effizienten Gestaltung auf Modellebene liegt.

4.2.3 Domänen

Jedes MLDesigner Modell, mit Ausnahme der Library, gehört einer Domäne an. Trotz dieser Zuordnung können Blockinstanzen unterschiedlicher Domänen durch den sogenannten Wormhole-Mechanismus miteinander kombiniert werden. MLDesigner unterstützt eine Vielzahl an Simulations- und Codegenerierungsdomänen. Folgende Übersicht erläutert die wichtigsten Simulationsdomänen:

- Die Domäne *Synchronous Data Flow* (SDF) wird zur Modellierung von datenflussorientierten Systemen genutzt, in denen Teilmodelle synchron eine fixe Anzahl von Datenpaketen austauschen. Hierbei ist die Ausführung von Teilmodellen lediglich Konsequenz des Vorhandenseins von hinreichend vielen Eingabedaten. Ein Zeitbegriff ist nicht vorhanden. Da die Produktions- und Konsumierungsraten zwischen den Teilmodellen statisch sind, ist der Kontrollfluss, das heißt die Ausführungsreihenfolge und -anzahl, voraussagbar und kann zu Simulationsbeginn ermittelt werden [Lee u. Messerschmitt 1987].

- Eine Erweiterung und als solche Obermenge zur SDF stellt die ebenfalls datenflussorientierte Domäne *Dynamic Data Flow* (DDF) dar. Im Gegensatz zur SDF sind hier die Produktions- und Konsumierungsraten zwischen den Teilmodellen variabel, woraus eine deutliche Erweiterung der Modellierungsfähigkeiten resultiert. Beispielsweise ermöglicht dies eine bedingte Ausführung und Rekursion von Teilmodellen. Jedoch folgt aus dem Verlust der statischen Ausführungsreihenfolge eine langsamere Simulationsgeschwindigkeit, da der Kontrollfluss für jeden Zyklus neu berechnet werden muss.

- Für den Entwurf zeitorientierter Modelle, wie Kommunikationsnetzwerke, digitaler Hardware oder Umgebungsmodelle technischer Systeme, eignet sich die *Discrete Event* (DE) Domäne. In diesem Berechnungsmodell ist die Semantik des Datenaustausches gleichzusetzen mit dem

Übermitteln von Ereignissen zu einer bestimmten Zeit. Der Simulator eines DE-Systems übermittelt Ereignisse chronologisch in der Reihenfolge ihres Auftretens. In einem DE-Modell liegt allen Teilmodellen die gleiche globale Zeitbasis zu Grunde.

- Die *Continous Time Discrete Event* (CTDE) Domäne unterstützt als weitere zeitbasierte Domäne zusätzlich zu diskreten Ereignissen einen kontinuierlichen Zeitverlauf. Dabei können Ein- und Ausgaben von Teilmodellen natürliche und kontinuierliche Werteverläufe zeigen. Somit stellt die CTDE-Domäne den natürlichsten Zusammenhang zur (makroskopischen) Realität dar. Ein CTDE-System ist eine alternative Darstellung eines Systems von Differentialgleichungen, beispielsweise zur Modellierung von physikalischen Prozessen.

- Interaktionsbeschreibungen mittels Zustandsautomaten werden durch die *Finite State Machine* (FSM) Domäne realisiert. Diese unterscheidet sich konzeptionell von den anderen MLDesigner Domänen, da sich die Semantik der Modelle nicht anhand verbundener Blockinstanzen ergibt. Die eigentliche Zustandsmaschine befindet sich innerhalb eines speziellen Primitives.

4.2.4 Datenfluss

In MLDesigner kommunizieren Blöcke mit ihrer Umgebung über Ports. Jeder Port besitzt einen Datentyp, durch den die Art der zu transferierenden Daten definiert wird, sowie eine Input-Output-Semantik. Relationen beschreiben die Verbindung zwischen mehreren Input- und Output-Ports und bestimmen so welche Blöcke miteinander Daten austauschen können. Blöcke die lediglich Output-Ports besitzen bezeichnet man als Quellen (engl.: Source). Umgekehrt nennt man Blöcke die nur Inputports besitzen Senken (engl.: Sink). MLDesigner unterscheidet weiterhin zwischen Single- und Multiports. Erstere können lediglich an einer Relation beteiligt sein, letztere an mehreren und ermöglichen die Definition einer variablen Anzahl von Ports gleichen Typs. In der aktuellen Version von MLDesigner ist die Verwendung von Multiports für Module und Primitives erlaubt, was jedoch für Module irrelevant ist, da diese zur Simulationszeit aufgelöst werden und der eigentliche Datenaustausch nur zwischen Primitives stattfindet. Zusätzlich zur direkten Kommunikation über Ports, ist auch eine indirekte, portlose Kommunikation anhand verlinkbarer Memory- Resource- und Event-Komponenten möglich.

MLDesigner unterstützt diverse Datentypen zum Nachrichtenaustausch zwi-

schen Blöcken. Neben Basistypen wie `Integer`, `Float`, `Complex`, `String` oder `Anytype` werden auch frei definierbare Datenstrukturen unterstützt. Dabei wird die Kompatibilität in Relation stehender Port-Datentypen zur Sicherstellung eines konsistenten Datentransports durch einen Modell-Prüfer analysiert. Der generische Datentyp `Anytype` nimmt in diesem Zusammenhang eine besondere Stellung ein, da er erst zur Simulationszeit anhand der verbundenen, typisierten Ports aufgelöst wird. Hervorzuheben ist, dass ein konsistenter Datentransport vor allem für die Softwareentwicklung von Bedeutung ist, da über 80% aller Fehlerursachen in diesem Bereich liegen [Salzwedel 2004b]. Folgende Datentyp-Untergruppen werden unterschieden:

- Kombinierte Typen
- Enumerationen
- Vektoren

Anzumerken ist, dass die Kompatibilität in Relation stehender Datenstrukturen lediglich auf der Verarbeitung durch die beteiligten Blöcke, sprich dem Lesen und Schreiben von Datenfeldern, basiert. Dies hat den Vorteil, dass mit Datenstrukturen typisierte Ports beliebig untereinander verbunden werden können, insofern nicht auf spezielle Datenfelder zugegriffen wird.

4.2.5 Simulation

Eine der Hauptaufgaben von MLDesigner liegt in der Simulation von Systemen, um deren Eigenschaften zu analysieren und Rückschlüsse auf die Realität zu ziehen. Hierzu können Modelle im Simulationsmodus zur Ausführung gebracht werden, wobei der Nutzer durch einen grafischen Debug- und Probe-Mechanismus unterstützt wird. Da Systeme aus mehreren Modellen mit unterschiedlichen Domänen bestehen, besitzt MLDesigner einen Multidomänen-Simulator. Dieser führt jedes Modell anhand seiner Domäne und dem aktuell zugeordneten Scheduler aus. Dabei setzt der Scheduler das domänenspezifische Berechnungsmodell um.

Definition 13: Ein *Scheduler* bestimmt und überwacht die strukturabhängige Ausführungsreihenfolge bzw. Blockinteraktion nach einem bestimmten mathematischen Formalismus.

Des weiteren besitzt jedes Modell ein sogenanntes Target, welches die Ausführung parametrisiert und koordiniert. Hierbei ist unter Ausführung nicht nur

Simulation zu verstehen, sondern beispielsweise kann dies auch eine Codegenerierung bedeuten. In diesem Fall manifestiert das Target eine Schnittstelle zu einer konkreten Implementation in Hard- oder Software. Da Targets eine gewisse Kontrolle über die Modellausführung erlauben, sind sie auch für Erweiterungen und Anpassungen interessant.

Definition 14: Ein *Target* koordiniert bei Modellausführung das Scheduling oder auch die Codegenerierung der durch Domänen beschriebenen Algorithmen.

Werden die domänenspezifischen Default-Targets verwendet, kommt es zur Modellausführung durch Simulation. MLDesigner unterscheidet generell zwischen interner und externer Simulation. Als intern bezeichnet man die direkte Simulation eines System im gleichen Adressraum der Oberfläche. Unter externer Simulation versteht man die Serialisierung des Systems in eine imperative Sprache und anschließendes Auslesen eines dedizierten Simulatorsystems in einen anderen Adressraum. Hierbei stehen die Varianten C++ und Ptolemy Tool Command Language (PTcl) zur Verfügung. Erstere ermöglicht den Schutz von geistigem Eigentum in Form von C++ Objektcode und die Verteilung von Simulationsreihen auf mehrere Rechner. Letztere zeichnet sich durch eine bessere Performance aus.

In diesem Zusammenhang unterstützt MLDesigner auch die Durchführung iterativer Simulationsdurchläufe. Hierzu werden Blöcken mehrere Kombinationen an Parameter-Belegungen zugeordnet oder auf Systemebene mehrere Parameter Sets definiert, woraus zur Ausführungszeit eine Menge an Simulationen resultiert. Eine weitere Möglichkeit stellt die Verwendung sogenannter Simulation Sets dar. Dabei handelt es sich um ausführbare Modelle, welche einen Ablaufplan zur Durchführung mehrerer, voneinander abhängiger Simulationen enthalten.

4.2.6 Adaptabilität

MLDesigner bietet mehrere Möglichkeiten zur Anpassung an konkrete Problemstellungen bzw. Projekte. Im folgenden sind diese nach ihrem Implementierungsaufwand aufsteigend sortiert:

- Erstellung spezifischer Modellbibliotheken
- Anbindung externen Quellcodes oder Programmaufrufe über Primitives

- Definition neuer Targets, Scheduler und Domänen
- Direkte Änderungen am Quellcode des MLDesigner

Die Erstellung spezifischer Modellbibliotheken ist direkt über die grafische Oberfläche von MLDesigner durchführbar. Dabei wird auch die Anbindung von externem Quellcode über die Primitive-Parameter `Linked Objects` und `Compile Options` unterstützt. Primitives erlauben zudem den Aufruf externer Programme. Beide Anpassungsmöglichkeiten sind recht schnell durchführbar und benötigen keine tiefgreifenden Kenntnisse über MLDesigner. Anders gestaltet sich die Definition und Anbindung neuer Targets, Scheduler und Domänen. Hier müssen die Klassen des Kernels von MLDesigner verwendet und erweitert werden. Die notwendige Integration erfolgt über eine vordefinierte Schnittstelle, welche bei Starten ausgewertet wird. Die letztgenannte Möglichkeit, das heißt die Durchführung direkter Änderungen am MLDesigner-Quellcode, ist nur in den wenigstens Fällen einsetzbar, da der Quellcode nicht frei verfügbar ist.

4.3 Verteilte Systeme

4.3.1 Charakteristik

Als verteiltes System bezeichnet man im Allgemeinen eine Menge interagierender Prozesse, die über keinen gemeinsamen Speicher verfügen und daher über Nachrichten miteinander kommunizieren. Eine Verteilung kann aus verschiedenen Gründen erforderlich bzw. vorteilhaft sein:

- Umsetzung verteilter Systemfunktionen
- Separation wiederverwendbarer Systemkomponenten
- Erhöhung der Ausfallsicherheit durch Verwendung redundanter Systemkomponenten
- Realisierung von Nebenläufigkeiten, respektive parallelen Prozessen, um beispielsweise Lasten zu verteilen
- Gemeinsame Nutzung einmalig vorhandener, kostenintensiver Systemkomponenten bzw. Betriebsmittel

Zur Beschreibung können unter anderem *Control Data Flow Graphs* (CDFG) verwendet werden [Platzner u. Thiele 2006]. Diese bestehen aus Knoten zur Erfassung von Funktionen, Tasks oder Prozessen, sowie aus gerichteten Kanten zur Darstellung von Abhängigkeiten und Kommunikationsverbindungen. Ein CDFG G ist als folgendes 2-Tupel definiert:

$$G = (V, E)$$

mit:

V = Menge an Knoten (engl.: Vertices)
E = Menge an Kanten (engl.: Edges)

Dabei beschreibt eine Kante $e \in E$ die Verbindung zwischen genau zwei Knoten und ist demzufolge ein 2-Tupel:

$$e = (v_s, v_e)$$

mit:

$v_s, v_e \in V$ = Start- und Endknoten

Abbildung 4.3 zeigt einen CDFG am Beispiel. Zu beachten ist, dass lediglich die Interaktionsbeziehungen zwischen Prozessen, unabhängig von ihrer Ausführungs- und Kommunikationsarchitektur erfasst werden. Da ohne diese eine Prozessausführung unmöglich ist, muss im Vorab eine Partitionierung durchgeführt werden. Dabei werden die Prozesse entsprechend ihren funktionalen sowie nichtfunktionalen Anforderungen auf eine Architektur verteilt (siehe 2.1). Bezogen auf den CDFG bedeutet dies, dass jeder Kante eine Verzögerung zugeordnet wird, welche von den zur Informationsübertragung verfügbaren Ressourcen abhängt. Charakteristisch für die Ausführungsarchitektur sind beispielsweise verfügbarer Speicher und CPU-Zeit, für die Kommunikationsarchitektur Art des Übertragungsmediums und Größe der Protokoll-Puffer. Die unterschiedlichen ressourcenabhängigen Verzögerungsarten lassen sich hierbei weiter verallgemeinern [Kurose u. Ross 2002]:

- Verarbeitungsverzögerung
- Warteschlangenverzögerung
- Übertragungsverzögerung
- Ausbreitungsverzögerung

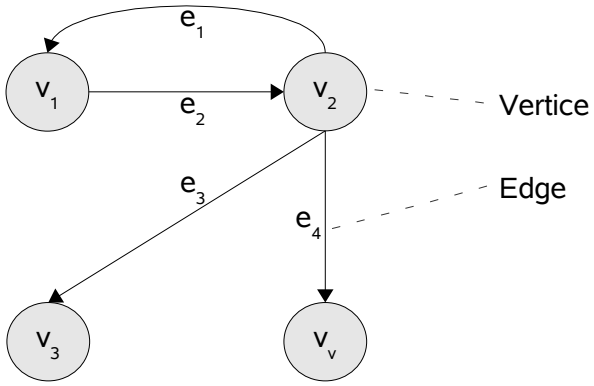

Abbildung 4.3: Verteilte Prozesse als Control Data Flow Graph

Fasst man diese zusammen erhält man die Gesamtverzögerung zur Informationsübertragung zwischen zwei Prozessen. Die Ermittlung, Abspeicherung und Auswertung solcher Verzögerungen ist eine der zentralen Aufgaben von Performance-Analysen. In diesem Zusammenhang ist anzumerken, dass auch andere Systemeigenschaften für Performance-Analysen relevant sind, wie beispielsweise Kosten, Energieverlust und Interferenzen.

4.3.2 Netzwerktopologien

Im Ergebnis der Systempartitionierung liegt eine bestimmte Anordnung von Architekturkomponenten, sowie physikalischen und logischen Verbindungen zur Informationsübertragung vor. Die hierdurch beschriebene Vernetzungsstruktur inklusive den verwendeten Übertragungsmedien und -protokollen wird als Netzwerktopologie bezeichnet. Zu beachten ist, dass vor der Erfassung zunächst ein Abstraktionsniveau festzulegen ist. Beispielsweise macht es einen Unterschied ob interagierende CPU-Komponenten oder vernetzte Rechner topologisch zu beschreiben sind. Dabei sind etwaig vorhandene Hardwarekomponenten mit Vermittlungsaufgaben, wie Switches und Hubs, ebenfalls zu berücksichtigen. Im weiteren Verlauf werden die in Abbildung 4.4 dargestellten Netzwerktopologien erläutert [Rech 2008]:

- Bei *Bus-Topologien* wird ein Übertragungsmedium durch mehrere Netzwerkteilnehmer gleichzeitig verwendet. Die Verbindungswege sind rela-

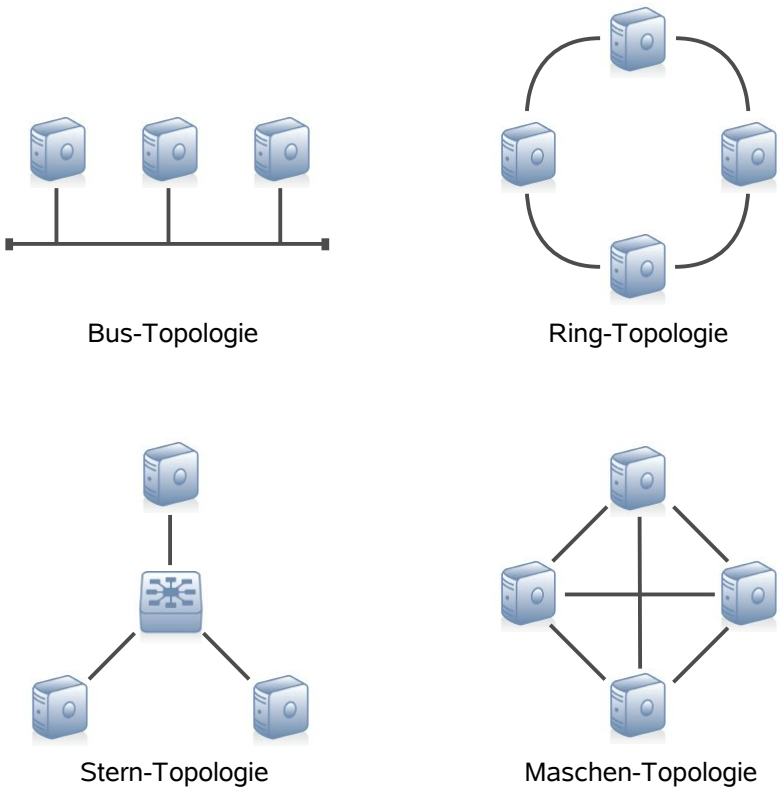

Abbildung 4.4: Netzwerktopologien im Überblick

tiv kurz, da im Normalfall eine direkte Kabelverlegung durchführbar ist. Zur Vermeidung von Reflexionen müssen beide Enden des Busses durch Widerstände terminiert werden. Bus-Topologien sind leicht erweiterbar, jedoch aufgrund der vieler Verbindungsstellen auch fehleranfällig. Außerdem ist eine Erweiterung während des Netzwerkbetriebs nicht möglich.

- Als *Ring-Topologie* bezeichnet man die Reihenschaltung mehrerer Netzwerkteilnehmer, wobei der letzte wieder mit dem ersten verbunden ist. Hieraus ergibt sich ein ununterbrochener Datenpfad, über den Informationen in eine Richtung übertragen werden können. Dies bedeutet gleichzeitig eine hohe Fehleranfälligkeit, da bereits eine Störung innerhalb des Rings zum Totalausfall führt. Die Fehlersuche gestaltet sich

jedoch als recht einfach, da infolge der Senderichtung lediglich der erste nicht empfangene Netzwerkteilnehmer ermittelt werden muss. Anzumerken ist, dass im Vergleich zur Bus-Topologie mehr Kabel benötigt wird, da der ringschließende Abschnitt hinzukommt. Eine Erweiterungen während des Netzwerkbetriebs ist auch hier unmöglich.

- Die *Stern-Topologie* verbindet mehrere Netzwerkteilnehmer sternförmig über einen zentralen Verteiler und ermöglicht eine sogenannte strukturierte Verkabelung. Sie ist vom Kabelbedarf aufwändiger als die bisher genannten Netzwerktopologien, bietet jedoch den Vorteil, dass sich Fehler lediglich auf eine Verbindung auswirken. Auch können diese recht einfach aufgefunden werden. Des weiteren ermöglicht sie eine dynamische Einbindung neuer Netzwerkteilnehmer während des Betriebs. Oftmals wird die Kombination mehrerer Stern-Topologien auch als *Baum-Topologie* bezeichnet.

- Innerhalb einer *Maschen-Topologie* besitzen alle Netzwerkteilnehmer direkte Verbindungen untereinander. Hieraus resultiert eine geringe Bandbreitennutzung, da die meisten Daten nur zu einzelnen Netzwerkteilnehmern übertragen werden müssen. Zudem beeinflussen Störungen stets nur eine Verbindung und können schnell ermittelt werden. Eine Erweiterung während des Netzwerkbetriebs ist ebenfalls unmöglich. Maschen-Topologien werden aus Kostengründen kaum für physikalische Verbindungen, sondern eher zur Realisierung logischer Verbindungen zwischen Routern verwendet.

Die Steuerung der Informationsübertragung zwischen Netzwerkteilnehmern beruht auf Protokollen. Hierbei handelt es sich beispielsweise um Mechanismen zur Adressierung, Synchronisation und Flusskontrolle. Nähere Informationen zum Aufbau und zur Spezifikation von Protokollen werden im anschließenden Abschnitt erläutert.

4.3.3 ISO/OSI Referenzmodell

Das Open Systems Interconnection (OSI) Referenzmodell der International Organization for Standardization (ISO) beschreibt allgemeine Prinzipien und Konstruktionsregeln zur abstrakten Spezifikation von Kommunikationsprotokollen. Hierbei werden sieben aufeinander aufbauende Schichten unterschieden, welche jeweils spezifische Kommunikationsaufgaben realisieren. Da das

Modell eine flexible Umsetzung dieser Schichten erlaubt, existiert mittlerweile eine Vielzahl unterschiedlicher Protokollspezifikationen. Folgende Grundgedanken führten zur Entwicklung des Referenzmodells und dessen Sieben-Schichten-Struktur [Tannenbaum 2003]:

- Eine neue Schicht ist immer dann erforderlich, wenn auch ein neuer Abstraktionsgrad erforderlich ist.
- Jede Schicht muss eine eindeutig definierte Funktion erfüllen.
- Bei der Funktionswahl sollte die Definition international genormter Protokolle berücksichtigt werden.
- Die Grenzen zwischen den Schichten sollten so gewählt werden, dass der Informationsfluss über deren Schnittstellen möglichst gering ist.
- Die Anzahl der Schichten sollte so groß sein, dass unterschiedliche Funktionen nicht in einer Schicht zusammengefasst werden müssen, aber so klein, dass die gesamte Architektur nicht unhandlich wird.

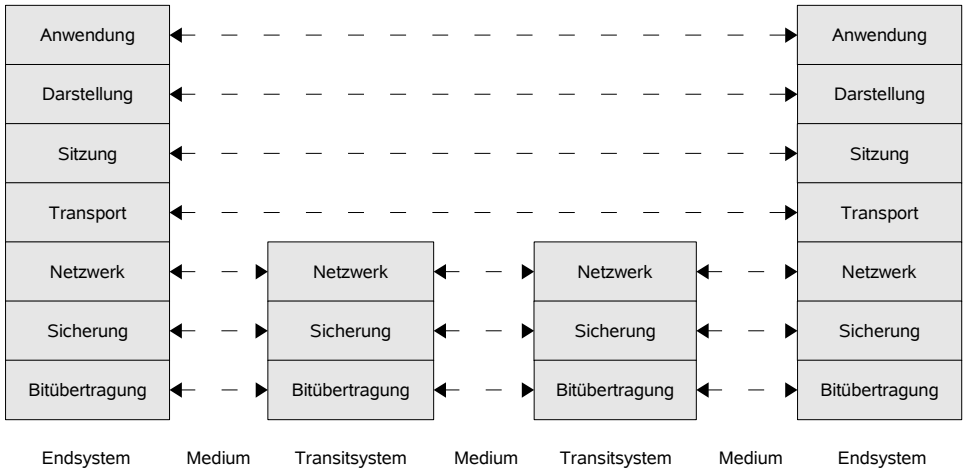

Abbildung 4.5: ISO/OSI Referenzmodell am Beispiel

Abbildung 4.5 zeigt die daraus abgeleitete Schichten-Struktur des ISO/OSI Referenzmodells am Beispiel zweier Endsysteme, die durch zwei Transitsysteme miteinander gekoppelt sind. Des weiteren ist zu erkennen, dass beim Entwurf nicht alle Schichten berücksichtigt bzw. verwendet werden müssen. Dies

hängt von der jeweilig umzusetzenden Funktionalität des Kommunikationsprotokolls ab. Beispielsweise muss ein Ethernet-Switch lediglich ankommende Daten zielgerichtet weiterleiten, weswegen keine anwendungsbezogene Schichten notwendig sind. Im folgenden werden die einzelnen Schichten aufgeführt und erläutert:

- **Schicht 1 – Bitübertragungsschicht – Physical Layer:**
  In dieser Schicht werden mechanische, elektrische und prozedurale Eigenschaften von physikalischen Verbindungen zur Bitübertragung zwischen Instanzen von Schicht 2 beschrieben. Instanzen von Schicht 1 sind stets über ein Übertragungsmedium miteinander verbunden, wie beispielsweise Lichtwellenleiter, Kupferkabel oder Funk.

- **Schicht 2 – Sicherungsschicht – Data Link Layer:**
  Aufgabe von Schicht 2 ist der Aufbau einer gesicherter Verbindung zwischen Instanzen von Schicht 3. Dazu gehören Verbindungsmanagement, Fehlererkennung und -korrektur, Reaktion auf nicht selbst korrigierbare Fehler sowie die Flusssteuerung mit dem Ziel einer bestmöglichen Ressourcenausnutzung. In Shared-Medium-Netzen, wie beispielsweise Ethernet, koordiniert Schicht 2 den Zugriff auf das gemeinsame Übertragungsmedium.

- **Schicht 3 – Netzwerkschicht – Network Layer:**
  In der Netzwerkschicht werden Endsysteme über Transitsysteme und Netze hinweg adressiert. Hierdurch können verbindungsorientierte Dienste eine virtuelle Ende-zu-Ende-Verbindung aufbauen und über diese kommunizieren. Wesentliche Funktionen sind das Routing und die Absicherung der Verbindungsqualität bzw. Quality of Service (QoS).

- **Schicht 4 – Transportschicht – Transport Layer:**
  Durch die Transportschicht wird die Differenz zwischen QoS-Parametern der nicht beeinflussbaren Transitsysteme und QoS-Forderungen der Anwendungsschicht ausgeglichen. Dies wird durch einen ordnungsgemäßen Verbindungsaufbau und -abbau, die Synchronisation der Verbindung, sowie eine Verteilung der Daten auf mehrere Verbindungen erreicht.

- **Schicht 5 – Sitzungsschicht – Session Layer:**
  Diese Schicht ermöglicht Sitzungen zwischen Anwendern und Programmen verschiedener Endsysteme. Dabei sind neben der Datenübertragung zusätzliche Organisations- und Synchronisationsdienste verfügbar. Beispielsweise kann festgelegt werden, ob ein Datentransfer in beide Richtungen oder lediglich in eine erfolgen darf. Im letzteren Fall regelt

die Sitzungsschicht die Zugriffsreihenfolge der Kommunikationspartner. Des weiteren können kurzfristige Verbindungsabbrüche behoben werden. Hierzu werden Synchronisationspunkte determiniert, an denen Sitzungen im Störungsfall fortgesetzt werden können. Ein nochmaliges Senden aller Daten wird vermieden.

- **Schicht 6 – Darstellungsschicht – Presentation Layer:**
  Durch die Darstellungsschicht wird die systemabhängige Darstellung von Daten in eine systemunabhängige transformiert. Dies erlaubt einen syntaktisch und semantisch korrekten Datenaustausch zwischen heterogenen Systemen. Weitere Aufgaben der Darstellungsschicht sind die Kompression und Verschlüsselung von Daten.

- **Schicht 7 – Anwendungsschicht – Application Layer:**
  Die Anwendungsschicht bildet die oberste Schicht der Modellhierarchie. Sie stellt Funktionen für Anwendungen zur Verfügung und ermöglicht die Dateneingabe und -ausgabe. Der eigentliche Anwendungsprozess liegt oberhalb der Schicht und ist nicht Bestandteil des Modells.

Neben der Strukturierung in einzelne Schichten erlaubt das Referenzmodell die Dekomposition in Teilschichten [Spaniol u. Jakobs 1993], was eine weitergehende Aufteilung an Funktionalität ermöglicht. Hieraus ergibt sich der Vorteil, dass Modifikationen innerhalb einer Schicht vorgenommen werden können, ohne die benachbarten Schichten zu beeinflussen.

Bezogen auf den Systementwurf und den Top-Down-Entwurfsansatz des Mission Level Designs sind die strukturellen Eigenschaften des Referenzmodells von besonderem Vorteil. Vor allem hinsichtlich der Wiederverwendbarkeit und Austauschbarkeit der Schichten, da diese durch standardisierte Modellkomponenten repräsentiert werden können. Voraussetzung ist die Definition kompatibler Schnittstellen, wie beispielsweise bereits in den Modellen der MLDesigner-Bibliothek `NetworkBuildingSet` gezeigt wurde. Dabei müssen die Modellkomponenten über eine einheitliche Ein- und Ausgabesemantik, sowie einen weitestgehend kompatiblen Daten- und Kontrollfluss verfügen. Nur so ist eine Ausführung und damit Validierung entworfener Kommunikationsprotokolle möglich. Zusammenfassend stellt das Referenzmodell einen wichtigen Ausgangspunkt für die im Rahmen der Arbeit definierte standardisierte Architekturbeschreibung dar.

4.3.4 Verteilungsdiagramme

Verteilungsdiagramme beschreiben die Ausführung- und Kommunikationsarchitektur von Systemen. Sie sind grundlegender Bestandteile der UML und bestehen aus mehreren Modellelementen [OMG 2004]. *Knoten* stellen existierende Ausführungkomponenten und *Kommunikationspfade* die Verbindungen zwischen diesen dar. Jeder Knoten kann wiederum verfeinert werden, was eine hierarchische Architekturmodellierung erlaubt. Des weiteren wird zwischen zwei speziellen Stereotypisierung von Knoten unterschieden. Bei *Geräteknoten* handelt es sich um zur Laufzeit physisch vorhandene Einheiten, die über Rechenleistung und Speicher verfügen. Unter *Ausführungsumgebungen* versteht man eine Menge von Diensten, die von den zugeordneten Verhaltensbeschreibungen zur Ausführungszeit benötigt werden. Ein solcher Knoten macht demzufolge nur innerhalb eines Geräteknotens Sinn. Anzumerken ist, dass Kommunikationspfade zum Austausch von Signalen und Nachrichten genau zwei Knoten verbinden. Etwaig definierte Multiplizitäten an den Enden der Kommunikationspfade beziehen sich auf die Anzahl von Knoten die angeschlossen werden könnten, wobei der Kommunikationspfad dann ebenfalls physisch mehrfach existieren muss.

Die Zuordnung von Systembestandteilen der Verhaltensbeschreibung auf Ausführungkomponenten wird durch *Artefakte* und *Verteilungsrelationen* beschrieben. Unter Artefakten versteht man physische Informationseinheiten, welche während des Entwicklungsprozesses aus Klassen und deren Verhaltensbeschreibungen erzeugt werden und somit das Verhalten manifestieren. Zur Darstellung dieses Zusammenhangs werden sogenannte *Manifestierungsrelationen* verwendet. Daraufhin können die erfassten Artefakte durch Verteilungsrelationen auf Knoten aufgeteilt werden. Damit stellen diese die eigentliche Verbindung zwischen Verhaltens- und Architekturmodell her. Da Knoten unterschiedliche Laufzeitverhalten besitzen können, bietet die UML mit *Verteilungsspezifikationen* die Möglichkeit zugeordnete Verhaltensbeschreibungen durch Ausführungsparameter zu adaptieren. Abbildung 4.6 stellt die Modellelemente und deren Beziehungen untereinander dar.

Verteilungsdiagramme erlauben die strukturelle Darstellung verteilter Systeme, sowie die Zuordnung bzw. Partitionierung von Funktions- auf Architekturkomponenten. Dabei erweist sich die visuelle Erfassung einiger Netzwerktopologien (siehe 4.3.2) als schwierig, da Kommunikationspfade lediglich zwischen genau zwei Knoten existieren dürfen. Auch können Kommunikationsprotokolle nur indirekt unter Verwendung architekturspezifischer Verhaltensbeschreibungen abgebildet werden. Als weiteres Manko ist die feh-

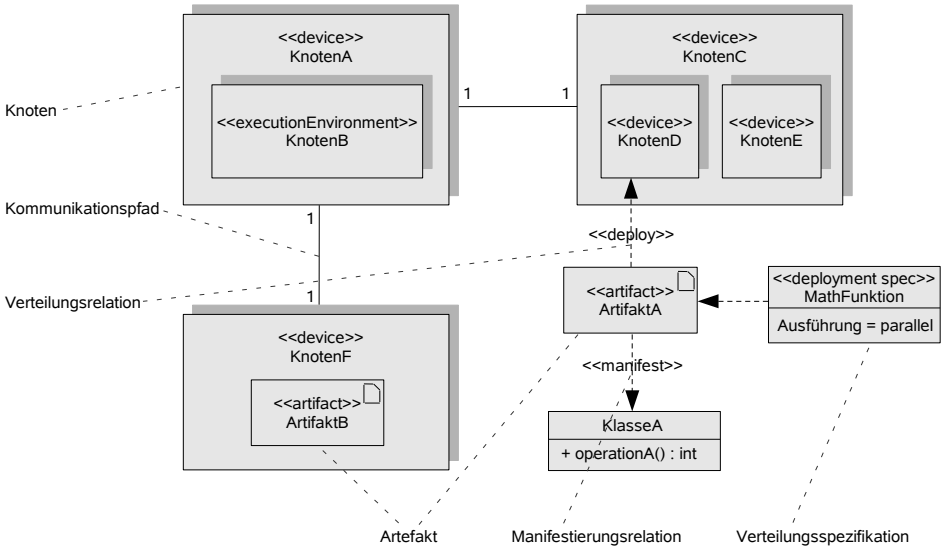

Abbildung 4.6: Verteilungsdiagramm

lende Möglichkeit zur Modellausführung zu benennen, was durch Anwendung und Erweiterung des UML-Metamodells jedoch prinzipiell erreicht werden kann. Ein Beispiel in Bezug auf Kompositionsstrukturdiagramme ist in [Grüner 2006] zu finden. Aufgrund der Strukturierung in Ausführungs- und Kommunikationskomponenten stellen Verteilungsdiagramme eine wichtige Grundlage zur Definition der weiter unten eingeführten standardisierten Architekturbeschreibung dar.

4.4 Konzeptentwicklung

Im folgenden werden Möglichkeiten zur Umsetzung der in Kapitel 3 vorgestellten Erweiterungen des Mission Level Designs analysiert. Vor- und Nachteile verschiedener Ansätze werden abgewogen, um letztendlich ein konkretes Konzept abzuleiten. Dabei wird Bezug auf die oben eingeführten und analysierten Grundprinzipien bzw. Vorarbeiten genommen.

Das erweiterte Mission Level Design ist durch die Einführung einer neuen Entwurfsebene gekennzeichnet, durch welche Funktion und Architektur assoziiert und iterativ betrachtet werden kann. Diese wird als *Electronic System*

Level bezeichnet und dient als Grundlage zur gemeinsamen Performance-Analyse, zur Iteration über Partitionierungen sowie zur plattformspezifischen Synthese. Eine Umsetzung kann nur durch Definition eines *standardisierten, ausführbaren Architekturmodells* erreicht werden, um darauf aufbauend die Realisierung eines synchronisierten Logging-Mechanismus zur Analyse, eines Modell-Generators zur Iteration, sowie einer Konfigurationsmöglichkeit zur Steuerung der Synthese zu ermöglichen. Hierbei bietet sich der Einsatz des Systementwurfswerkzeugs MLDesigner an, da dort bereits eine Reihe umsetzungsrelevanter Features vorliegen. Zudem können Erweiterungen relativ leicht durchgeführt werden. Im weiteren Verlauf wird MLDesigner bezüglich der Umsetzung von Eigenschaften verteilter Systeme, wie Netzwerktopologien und Protokollen, analysiert.

Zur Erfassung der Topologie verteilter Systeme können Verteilungsdiagramme verwendet werden. Eine solche Beschreibung ist ebenfalls in Form von MLDesigner-Modellelementen möglich. Hierbei entsprechen Knoten Modulen und Kommunikationspfade Relationen, wobei erstere eine standardisierte Port-Schnittstelle zur Interaktion mit benachbarten Knoten besitzen müssen. Auch die Erfassung mehrstufiger Topologien ist aufgrund des hierarchischen Modellierungskonzepts von MLDesigner problemlos durchführbar. Des weiteren müssen auf jeden Knoten die zu berechnenden Funktionen verteilt bzw. partitioniert werden, weswegen diese im weiteren Verlauf auch als Partitionen bezeichnet werden.

Um spezifizierte oder am realen System ermittelte Performance-Eigenschaften in das Modell zu integrieren, muss die erläuterte Grundidee erweitert werden. In Bezug auf Kommunikationspfade bedeutet dies, dass zur Verhaltensbeschreibung der Bitübertragungsschicht des ISO/OSI Referenzmodells ein zusätzliches Modul eingeführt werden muss, da MLDesigner-Relationen kein Verhalten zugeordnet werden kann. Dieses Modul wird als Kanal bezeichnet und benötigt ebenfalls eine standardisierte Port-Schnittstelle. Konkret sind Multiports zu verwenden, da Kanäle mit mehreren Partitionen verbunden sein können und nur so eine Adressierung möglich ist.

Analog zu Kommunikationspfaden besitzen auch Partitionen Performance-Eigenschaften, die durch zusätzliche Modellelemente zu erfassen sind. Unter anderem muss das Verhalten der restlichen zur Kommunikation notwendigen Schichten des ISO/OSI Referenzmodells beschrieben werden. Hierzu wird ein Modul namens Protokollstapel eingeführt. Durch dieses werden die einzelnen Protokolle anhand der Schichten sowie der Pack-Entpack-Semantik des ISO/OSI Referenzmodells erfasst, wobei wiederum eine standardisierte Port-Schnittstelle notwendig ist. Insofern die Schnittstelle eingehalten wird,

können auch nicht ISO/OSI-konforme Kommunikationsprotokolle beschrieben werden. Da mehrere partitionierte Funktionen mit dem Protokollstapel verbunden sein können und demzufolge zu adressieren sind, besteht dessen interne Schnittstelle aus Multiports. Hierbei ist zu beachten, dass durch die eingefügten Architekturkomponenten die funktionale Struktur aufgelöst wird, welche jedoch Voraussetzung zur Modellausführung ist. Eine Möglichkeit zur Erfassung sind die in Abbildung 4.7 erläuterten Mapping-Einträge unter Verwendung voll qualifizierter Block-Port-Namen. Um Mapping-Einträge global verwalten zu können, bietet sich die Definition eines neuen MLDesigner-Targets an. Anzumerken ist, dass im Rahmen der Konzeptentwicklung auch alternative Ansätze untersucht wurden. Unter anderem wurden Protokollstapel mit in Kanäle integriert, sprich alle Schichten des ISO/OSI Referenzmodells dort erfasst. Da jedoch einerseits nicht jede mit dem Kanal verbundene Partition den gleichen Protokollstapel verwenden muss, wie beispielsweise bei der Kommunikation von Computern über einen Switch, und andererseits Protokollstapel Einfluss auf die Performance der Partitionen haben, müssen diese den Partitionen zugeordnet werden.

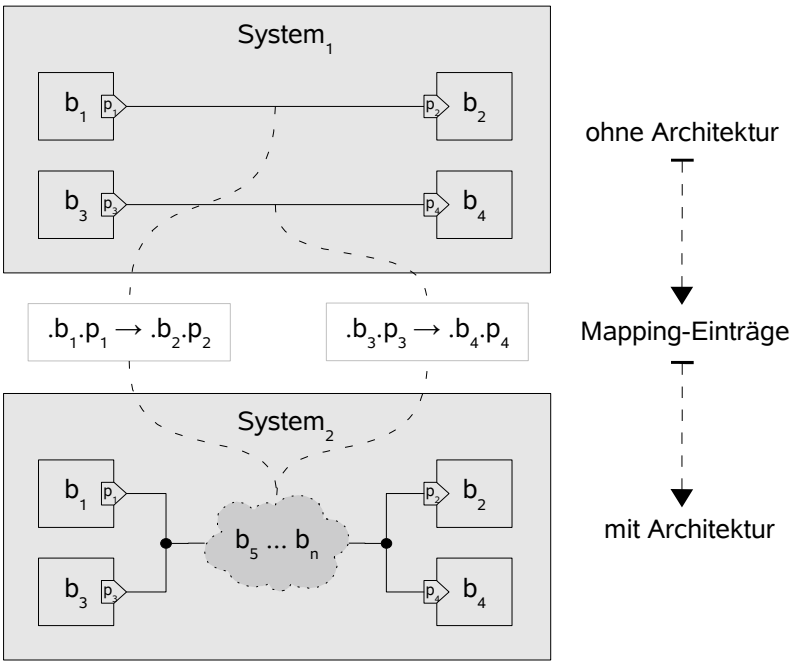

Abbildung 4.7: Ableitung von Mapping-Einträgen

Zusätzlich zur Kommunikations-Performance besitzen Partitionen auch eine Ausführungs-Performance. Um diese zu erfassen, kann die in [Zens 2003] und [Paluch 2004] erläuterte Grundidee erweitert werden. Hierbei wird die Ausführung der im Datenflussmodell beschriebenen Funktionen in einem standardisierten Server zentralisiert. Zur Umsetzung ist ein neues Modul notwendig, welches als Ausführungseinheit bezeichnet wird und wie Protokollstapel über eine standardisierte Port-Schnittstelle mit Multiports verfügen muss. Dessen Aufgabe ist es, die Ausführung der zugeordneten Funktionen in Abhängigkeit der verfügbaren Ressourcen zu steuern. Zuvor müssen alle Funktionen mit der Ausführungseinheit verbunden werden, wobei eine sofortige Integration durch den Nutzer unzweckmäßig ist, da die in Form von Relationen beschriebene funktionale Modellstruktur während der Modellerstellung nicht mehr erkennbar wäre. Demzufolge ist eine temporäre Integration zur Simulationszeit zu bevorzugen. Dies wird durch die bisherigen auf Modellbibliotheken basierenden Ansätze nicht unterstützt. Um dennoch eine automatisierte Verbindung der Funktionsblöcke mit der Ausführungseinheit zu Simulationsbeginn zu ermöglichen, bietet sich der Einsatz eines speziell angepassten MLDesigner-Targets an. Die Art der Ausführungseinheit ist dabei über einen Partitions-Parameter festzulegen.

Hervorzuheben ist, dass durch die genannten Architekturkomponenten der Entwurfsprozess vereinfacht und beschleunigt werden kann, da nunmehr eine Abgrenzung zwischen Funktion und Architektur möglich ist, was zudem einen problemlosen Austausch erlaubt. Hierzu müssen während des Entwurfsprozesses die standardisierten Schnittstellen eingehalten werden. Die Verhaltensbeschreibung an sich kann jedoch auf einem beliebigen Abstraktionsniveau vollzogen werden, da es beispielsweise nicht immer erforderlich ist eine Analyse auf Bitebene durchzuführen. Zudem sei an dieser Stelle erwähnt, dass aus den strukturellen Modellabänderungen mehrere Probleme folgen. Diese werden unter anderem in Abschnitt 4.9 erläutert.

Basierend auf den standardisierten Architekturkomponenten kann nun ein Konzept zur gemeinsamen Performance-Analyse abgeleitet werden. Ohne einen Standard ist dies nicht ohne weiteres realisierbar, da ein gemeinsamer Ansatzpunkt zur Datenentnahme fehlt. Zunächst ist abzuklären, wie die notwendigen Messungen durchgeführt werden können. Hierzu bietet sich der Einsatz der Simulationsmöglichkeiten von MLDesigner an, konkret der sogenannten DE-Domäne, da ressourcenabhängige Verzögerungen zu untersuchen sind. Dies hat den Vorteil, dass Verzögerungen direkt auf die Scheduler-Zeit der DE-Domäne abgebildet werden können. Voraussetzung zur Ausführung ist die Erfassung entsprechender Missionen und Funktionen, wobei die

Multidomänen-Fähigkeiten von MLDesigner zum Tragen kommen. Dabei muss der Datenfluss zwischen den Architekturkomponenten durch standardisierte Datenstrukturen beschrieben werden. Nur so kann die Kompatibilität und Austauschbarkeit der Architekturkomponenten untereinander sichergestellt, sowie eine parallele Übertragung von Daten und analyserelevanten Zusatzinformationen ermöglicht werden. Des weiteren ist festzulegen, wie die simulativ ermittelten Messwerte synchronisiert erfasst und ausgewertet werden können. In diesem Zusammenhang ist eine zentrale Erfassung einer dezentralen vorzuziehen, da die Datensätze zur weitergehenden Verarbeitung schneller verfügbar sind (z.B. zur Anbindung einer Datenbank oder eines externen Programms) und zudem mehrere Datensätze kombiniert ausgewertet werden können (z.B. Durchsatz-Matrix) [Liebezeit 2005]. Um dies zu ermöglichen, ist eine systemweit verfügbare Schnittstelle erforderlich, welche auf mehrere Arten realisiert werden kann (siehe auch Abschnitt 4.2.6). Eine Variante besteht darin, die Messwerte durch sogenannte MLDesigner-Probes zu sammeln. Diese können jedoch nur manuell durch den Nutzer hinzugefügt werden, weswegen keine vollständige Automatisierung möglich ist. Des weiteren können die Messwerte durch eine direkte Anbindung an MLDesigner erfasst werden. Da jedoch kein Zugriff auf den Quellcode besteht, ist diese Option auszuschließen. Auch ist eine Erfassung durch ein externes Programm möglich, was jedoch einen zusätzlichen Verwaltungs-Overhead nach sich zieht. Letztlich ist eine Anbindung in Form eines MLDesigner-Targets zu bevorzugen. Targets besitzen eine direkte Schnittstelle zu den Modellelementen die sie beinhalten und können demzufolge bei Auswahl auf oberster Hierarchieebene alle Messwerte systemweit erfassen. Zusätzlich bieten sie die Möglichkeit zu Simulationsbeginn modellbezogene Aktionen durchzuführen, wie weiter oben bereits erläutert.

Um für ein Funktionsmodell eine geeignete Architektur zu ermitteln, muss eine Möglichkeit zur iterativen Analyse und Bewertung von Architekturalternativen geschaffen werden. Der hierfür notwendige iterative Prozess kann prinzipiell anhand eines Simulation Sets erfasst und ausgeführt werden. Jedoch fehlt zur vollständigen Automatisierung ein Generator, welcher auf Grundlage des Funktionsmodells und standardisiert beschriebener Architekturkomponenten Architekturmodelle erzeugt. Die Realisierung eines solchen Generators ist einerseits durch eine direkte Anbindung an die intere Klassenstruktur von MLDesigner und andererseits in Form einer externen Anwendung auf Grundlage der XML-basierten Modellbeschreibungen möglich. Dabei bietet eine direkte Anbindung den Vorteil eines geringeren Implementationsaufwandes, setzt jedoch Zugriffsrechte auf den MLDesigner-Quellcode voraus. Da diese im Rahmen der Arbeit nicht bestehen, ist die zweite Varian-

te zu bevorzugen. Dies bedeutet, dass Modelle durch einen Parser eingelesen und verarbeitet werden müssen, um letztlich ein neues Modell zu generieren. Hierbei sind bereits im Vorab feststehende Architekturkomponenten zu berücksichtigen.

Auf Grundlage der standardisierten Architekturbeschreibung und der gewählten Partitionierung ist abzuklären, wie eine plattformspezifische Synthese durchgeführt werden kann. Ziel ist es, die Funktionen unter Berücksichtigung der gewählten Architektur in eine Implementation zu transformieren. Hierzu müssen die Funktionen in Form einer synthetisierbaren Sprache vorliegen. Eine Möglichkeit bieten SDF-basierte Blockflussdiagramme in Verbindung mit eingebetteten Zustandsmaschinen [Rath u. Salzwedel 2004]. Zur Synthese der Protokollstapel ist es hierbei von Vorteil bereits vordefinierte IP-Cores zu verwenden, wie beispielsweise auf Basis der Wishbone-Spezifikation [Open 2002], da diese bereits validiert und zudem wiederverwendbar sind. Im Ergebnis entsteht ein Netzwerk interagierender IP-Cores. Da die Synthese durch die Überführung einer generischen in eine speziellere Beschreibungsform gekennzeichnet ist, müssen implementationsspezifische Zusatzinformationen unter Verwendung einer Konfigurationssprache in das Modell integriert werden. Die standardisierten Architekturkomponenten dienen hierbei als vereinheitlichte Konfigurationsschnittstelle. Zur Umsetzung bieten sich zwei Varianten an. Einerseits ist eine direkte Anbindung an MLDesigner denkbar, andererseits eine iterative Transformation der XML-basierten Modellbeschreibungen. Da kein Zugriff auf den Quellcode von MLDesigner besteht, ist letztere Variante zu bevorzugen. Die Verarbeitung von XML kann wiederum anhand von XSLT-Skripten oder eines XML-Parsers durchgeführt werden. Aufgrund der Tatsache, dass während der Synthetisierung keine strukturellen Änderungen am Modell notwendig sind, ist der Einsatz von XSLT-Skripten günstiger. Zudem ist der Implementierungsaufwand geringer.

Im weiteren Verlauf muss das vorgeschlagene Konzept konkretisiert werden. Hierzu ist zunächst eine Formalisierung der standardisierten Architekturbeschreibung durchzuführen, um anschließend ein Anwendungsumgebung zur Realisierung der geforderten Features abzuleiten. Daraufhin müssen Anwendungsbeispiele zur Validierung entworfen und analysiert werden. Folgende Übersicht fasst die Grundideen des Konzepts in Bezug auf eine Umsetzung mittels MLDesigner zusammen:

- Standardisierte Erfassung hierarchischer Netzwerktopologien unter Berücksichtigung der Verteilungsdiagramm-Notation
- Verwendung von Ports bzw. Multiports zur statischen, sowie von Daten-

strukturen zur dynamischen Definition standardisierter Schnittstellen

- Standardisierte Erfassung von Kommunikationsprotokollen unter Berücksichtigung der Schichten des ISO/OSI Referenzmodells
- Realisierung der Pack-Entpack-Semantik des ISO/OSI Referenzmodells durch kompatible, verschachtelbare Datenstrukturen
- Einsatz von Multiports zur Adressierung von Datenpaketen
- Definition eines neuen Targets zur Durchführung zentraler, systemübergreifender Konfigurations- und Initialisierungsaufgaben vor der Simulation, sowie von Analyseaufgaben während der Simulation
- Adressierung architekturbedingt getrennter Funktionen unter Verwendung von Mapping-Einträgen
- Dynamische Einbindung von Ausführungseinheiten, respektive Blöcken, zur Simulationszeit
- Verwendung der DE-Scheduler-Zeit zur Erfassung und Analyse von Performance-Eigenschaften
- Kombination unterschiedlicher Domänen zur Beschreibung von Mission, Funktion und Architektur
- Generierung von Architekturmodellen unter Verwendung eines XML-Parsers und einer Konfigurationssprache
- Synthetisierung von IP-Cores unter Einsatz von XSLT und einer Konfigurationssprache

4.5 Formalisierung

Basierend auf der Konzeptentwicklung und der damit verbundenen Analyse von Eigenschaften, Standards und Entwurfsmöglichkeiten verteilter Systeme wird im weiteren Verlauf die abgeleitete standardisierte Architekturbeschreibung formalisiert. Diese ist zwingend notwendig um gemeinsame Modellelemente als Ausgangspunkt für vergleichbare frühzeitige Performance-Analysen, zur plattformspezifischen Synthese und zur manuellen oder automatisierten Iteration über Architektur zu definieren. Speziell hinsichtlich der automatisierten Architekturiteration und der damit verbundenen Generierung von Architekturmodellen sind Modellelemente mit determinierten

Schnittstellen erforderlich. Nur so kann eine flexible Austauschbarkeit und Wiederverwendbarkeit, sowie die Einbindung benutzerdefinierter Architekturkomponenten sichergestellt werden. Anzumerken ist, dass die konkrete Implementierung der Modellelemente und etwaig notwendiger Werkzeuge zunächst nicht im Vordergrund steht. Eine gesonderte Betrachtung erfolgt zu einem späteren Zeitpunkt.

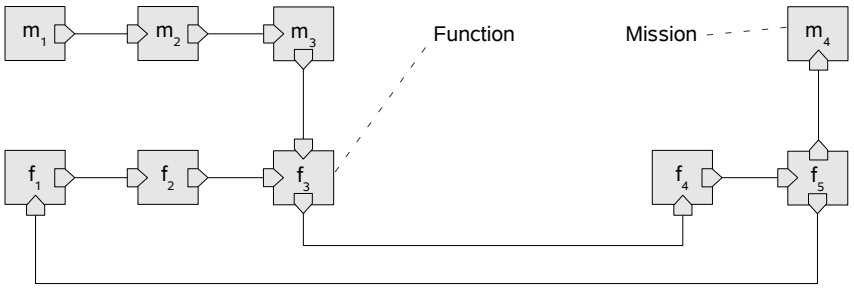

Abbildung 4.8: Funktionales Gesamtsystemmodell

Ausgangspunkt des in Abschnitt 3.3 erläuterten Entwurfsprozesses stellt ein funktionales Gesamtsystemmodell dar. Darunter versteht man ein integriertes Modell von Systemfunktion und -umgebung, in welchem die Umstände zur Erbringung der Funktion nicht berücksichtigt werden (siehe auch 2.1). Die Systemarchitektur wird demzufolge ausgeblendet. Das funktionale Gesamtsystemmodell wird ganzheitlich in Form einer ausführbaren Spezifikation beschrieben, wodurch eine simulative Validierung ermöglicht wird. Ziel ist es, die funktionalen Systemanforderungen durch Funktionskomponenten und die zur Validierung notwendige Umgebung, inklusive der analyserelevanten Anwendungsfälle, mittels Missionskomponenten zu erfassen. Abbildung 4.8 stellt die strukturellen Zusammenhänge am Beispiel dar. Ein *funktionales Gesamtsystemmodell* (engl.: Functional Common System Model) S_{Func} ist ein 2-Tupel:

$$S_{Func} = (M, F)$$

mit:

M = Menge an Missionskomponenten (engl.: Missions)
F = Menge an Funktionskomponenten (engl.: Functions)

Zur Abbildung nichtfunktionaler Anforderungen muss das funktionale Gesamtsystemmodell um Architekturkomponenten erweitert werden. Das resultierende ganzheitliche Modell wird im weiteren Verlauf als architekturelles Gesamtsystemmodell bezeichnet und liegt in Form einer ausführbaren Spezifikation vor. Es besteht aus Missionskomponenten, Partitionen und Kanälen, wobei die Missionskomponenten denen des funktionalen Gesamtsystemmodells entsprechen. Dies gilt prinzipiell auch für die Funktionskomponenten,

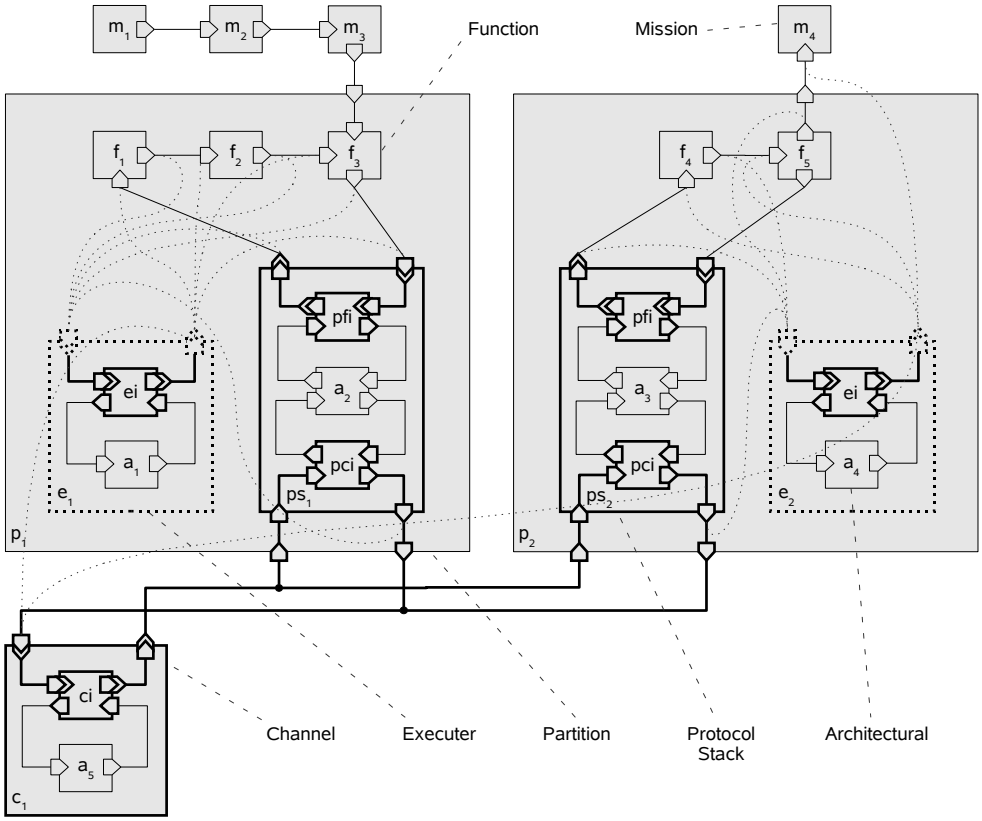

Abbildung 4.9: Architekturelles Gesamtsystemmodell

jedoch werden diese komplett auf unterschiedliche Partitionen bzw. Ausführungseinheiten aufgeteilt. Kanäle stellen dabei die Kommunikation zwischen Partitionen dar und sind Bestandteil einer erweiterbaren Bibliothek vorgefertigter Architekturkomponenten. Ziel des architekturellen Gesamtsystemmodells ist es, eine gemeinsame Validierung funktionaler und nichtfunktionaler Anforderungen zu ermöglichen. In Abbildung 4.9 wird es am Beispiel zweier

Partitionen dargestellt, die durch einen Kanal verbunden sind. Ein *architekturelles Gesamtsystemmodell* (engl.: Architectural Common System Model) S_{Arch} ist ein 3-Tupel:

$$S_{Arch} = (M, P, C)$$

mit:

M = Menge an Missionskomponenten
$P \subseteq P_{All}$ = Menge an Partitionen (engl.: Partitions)
$C \subset C_{ArchLib}$ = Menge an Kanälen (engl.: Channels)

Partitionen bestehen aus Funktionskomponenten, einer Ausführungseinheit, Protokollstapeln, sowie hierarchisch eingebetteten Partitionen und Kanälen. Aufgabe der Ausführungseinheit ist die ressourcenabhängige Ausführung aller Funktionen und Protokollstapel der Partition. Dabei werden zur modellbasierten Beschreibung der Performance-Eigenschaften quantitative Ressourcen (z.B. Speicher) und Server-Ressourcen (z.B. CPU-Prioritäten) eingesetzt. Im Ergebnis kommt es zur Verschiebung der Ausführungsreihenfolge und Verzögerung der Datenpakete. Die Kommunikation mit der Partitionsumgebung, sprich dem Umgebungsmodell und anderen Partitionen, verläuft über Protokollstapel. Diese übersetzen den funktionalen in einen architekturellen Datenstrom und zurück. Auch hier werden Performance-Eigenschaften mittels Ressourcen abgebildet und führen zu Verzögerungen. Zusätzlich werden die zu übertragenden Datenpakete entsprechend der verwendeten Protokolle fragmentiert bzw. defragmentiert. Alle Ausführungseinheiten, Protokollstapel und eingebettete Kanäle sind Bestandteil einer vorgefertigten, durch den Nutzer erweiterbaren Bibliothek an Architekturkomponenten. Des weiteren können Partitionen eingebettete Partitionen und Kanäle besitzen, durch welche das Partitionierungskonzept hierarchisch weitergeführt wird. Am Ende der hierarchischen Strukturierung stehen stets Funktionen und Protokollstapel. Eine *Partition* $p \in P$ ist ein 5-Tupel:

$$p = (F_{Part}, e, PS, P, C)$$

mit:

$F_{Part} \subseteq F$ = Menge an Funktionskomponenten
$e \in E_{ArchLib}$ = Ausführungseinheit (engl.: Executer)
$PS \subset PS_{ArchLib}$ = Menge an Protokollstapeln (engl.: Protocol-Stacks)
$P \subseteq P_{All} \setminus p$ = Menge an eingebetteten Partitionen
$C \subset C_{ArchLib}$ = Menge an eingebetteten Kanälen

Jede Partition besitzt genau eine Ausführungseinheit. Diese besteht aus einer standardisierten Ausführungseinheit-Schnittstelle sowie verschiedenen Architekturkomponenten. Erstere gewährleistet die Kompatibilität und Austauschbarkeit der Ausführungseinheit und ist Bestandteil der Basisbibliothek, letztere ermöglichen eine freie anwendungsbezogene Beschreibung architektureller Eigenschaften. Beispielsweise können Betriebssysteme entworfen werden, welche die partitionierten Funktionen unter Berücksichtigung verfügbarer Ressourcen als Tasks ausführen.

Definition 15: Unter einem *Task* versteht man eine in sich geschlossene Aufgabe, die durch einen Computer unabhängig von anderen Tasks abgearbeitet werden kann.

Anzumerken ist, dass auch mehrere CPU's simultan innerhalb einer Ausführungseinheit nachgebildet werden können. Eine *Ausführungseinheit* $e \in E_{ArchLib}$ ist ein 2-Tupel:

$$e = (A, ei_{BaseLib})$$

mit:

A = Menge an Architekturkomponenten (engl.: Architecturals)
$ei_{BaseLib}$ = Ausführungseinheit-Schnittstelle (engl.: Executer-Interface)

Wie bereits erwähnt, kommunizieren Partitionen über Protokollstapel mit ihrer Umgebung. Abbildung 4.9 stellt diese als Bestandteil der beiden Partitionen dar. Als Basis zur Definition des Protokollstapels als auch des weiter unten erläuterten Kanals, wurde aufgrund seiner Schichten-Struktur das ISO/OSI Referenzmodell gewählt. Prinzipiell können jedoch auch andere eingesetzt werden. Dabei ist zu beachten, dass es aus Effizienzgründen nicht zwingend notwendig ist alle Schichten zu modellieren, insofern diese nicht relevant sind. Protokollstapel bestehen aus einer standardisierten Protokollstapel-Funktion-Schnittstelle, verschiedenen protokollspezifischen Architekturkomponenten und einer standardisierten Protokollstapel-Kanal-Schnittstelle. Die Protokollstapel-Funktion-Schnittstelle gewährleistet die Kompatibilität und Austauschbarkeit in Richtung der zugeordneten Funktionen bzw. der ISO/OSI Anwendungsschicht. Durch die Protokollstapel-Kanal-Schnittstelle wird dies in Richtung des Kanals bzw. der ISO/OSI Bitübertragungsschicht sichergestellt. Beide Komponenten gehören zur Basisbibliothek. Zur Beschreibung des eigentlichen Protokollstapels bzw. der restlichen ISO/OSI Schichten werden frei definierbare Architekturkomponenten verwendet. Die dabei zu be-

schreibenden Protokolle und deren Ressourcen können auf einem beliebigen Abstraktionsniveau entworfen werden. Jedoch ist es von Vorteil auch zwischen den Protokollen eine standardisierte Schnittstelle zu verwenden, um die Wiederverwendung durch andere Protokollstapel zu ermöglichen. Diese basiert auf einer determinierten Port-Semantik in Verbindung mit kompatiblen Datenstrukturen. Da eine möglichst freie Gestaltung von Protokollstapeln ermöglicht werden soll, gehören die internen Protokollschnittstellen nicht zum formalisierten Modell. Eine Erläuterung hierzu folgt später. Ein *Protokollstapel* $ps \in PS_{ArchLib}$ ist ein 3-Tupel:

$$ps = (A, pfi_{BaseLib}, pci_{BaseLib})$$

mit:

A = Menge an Architekturkomponenten
$pfi_{BaseLib}$ = Protokollstapel-Funktion-Schnittstelle (engl.: Protocol-Function-Interface)
$pci_{BaseLib}$ = Protokollstapel-Kanal-Schnittstelle (engl.: Protocol-Channel-Interface)

Die Kommunikation zwischen Partitionen bzw. deren Protokollstapel basiert auf Kanälen. Diese repräsentieren das physikalische Übertragungsmedium, also die ISO/OSI Bitübertragungsschicht. Kanäle bestehen aus einer standardisierten Kanal-Schnittstelle und mehreren nutzerdefinierten Architekturkomponenten. Erstere gehören zur Basisbibliothek und gewährleisten die Kompatibilität und Austauschbarkeit untereinander, ähnlich den Schnittstellen der Ausführungseinheiten und Protokollstapel. Letztere erlauben die ressourcenabhängige Beschreibung architektureller Eigenschaften. Ein *Kanal* $c \in C_{ArchLib}$ ist ein 2-Tupel:

$$c = (A, ci_{BaseLib})$$

mit:

A = Menge an Architekturkomponenten
$ci_{BaseLib}$ = Kanal-Schnittstelle (engl.: Channel-Interface)

Damit ist die Definition des architekturellen Gesamtsystemmodells, sowie des als Ausgangspunkt dienenden funktionalen Gesamtsystemmodells abgeschlossen. Im weiteren Verlauf werden die erläuterten Modellelemente mittels Mengen kategorisiert, um letztlich die zentrale Bibliothek der Arbeit, das sogenannte Architecture Block Set, zu definieren. Die erste Menge wird als Basis-

bibliothek bezeichnet. Diese besteht aus allen Modellelementen, die zur Einhaltung der standardisierten Schnittstellen beim Entwurf neuer Architekturelemente notwendig sind. Dabei handelt es sich um die Ausführungseinheit-Schnittstelle, Protokollstapel-Funktion-Schnittstelle, Protokollstapel-Kanal-Schnittstelle und Kanal-Schnittstelle. Wie später gezeigt wird, beinhaltet die Basisbibliothek noch weitere Komponenten, welche jedoch hinsichtlich der Formalisierung irrelevant sind. Beispielsweise existieren mehrere Komponenten zur einheitlichen Erfassung von Analysedaten. Die *Basisbibliothek* $BaseLib$ ist die Vereinigungsmenge:

$$BaseLib = \{ei_{BaseLib} \cup pfi_{BaseLib} \cup pci_{BaseLib} \cup ci_{BaseLib}\}$$

mit:

$ei_{BaseLib}$ = Ausführungseinheit-Schnittstelle
$pfi_{BaseLib}$ = Protokollstapel-Funktion-Schnittstelle
$pci_{BaseLib}$ = Protokollstapel-Kanal-Schnittstelle
$ci_{BaseLib}$ = Kanal-Schnittstelle

Die zweite Menge fasst alle nach dem Standard entworfenen Architekturelemente zusammen und wird als Architekturbibliothek bezeichnet. Sie besteht aus Ausführungseinheiten, Protokollstapeln und Kanälen. Da die Elemente der jeweiligen Teilmengen zueinander kompatibel sind, ist ein Austausch zur Iteration über verschiedene Architekturalternativen problemlos möglich. Auch der Entwurf neuer Architekturelemente ist unter Verwendung der Basisbibliothek jederzeit durchführbar. Die *Architekturbibliothek* $ArchLib$ ist die Vereinigungsmenge:

$$ArchLib = \{E_{ArchLib} \cup PS_{ArchLib} \cup C_{ArchLib}\}$$

mit:

$E_{ArchLib}$ = Menge an Ausführungseinheiten
$PS_{ArchLib}$ = Menge an Protokollstapeln
$C_{ArchLib}$ = Menge an Kanälen

Beide Mengen definieren in Kombination mit dem im späteren Verlauf erläuterten ESL-Target und Architektur-Generator das Architecture Block Set. Dieses fasst alle im Rahmen der Arbeit entwickelten Modellelemente und Werkzeuge bzw. Anwendungen zusammen. Die anschließenden Abschnitte beschäftigen sich mit der Implementation des Architecture Block Sets in Form einer Anwendungsumgebung für das Systementwurfswerkzeug MLDesigner.

Das *Architecture Block Set ABS* ist ein 4-Tupel:

$$ABS = (BaseLib, ArchLib, ESLTarget, ArchGen)$$

mit:

$BaseLib$ = Basisbibliothek
$ArchLib$ = Architekturbibliothek
$ESLTarget$ = ESL-Target
$ArchGen$ = Architektur-Generator

4.6 Realisierung der Anwendungsumgebung

4.6.1 Einführung

Die Umsetzung des erweiterten Mission Level Designs wird durch die Definition einer mehrteiligen Anwendungsumgebung für das Systementwurfswerkzeug MLDesigner erreicht. Hierbei werden alle notwendigen Modellelemente und Werkzeuge bzw. Anwendungen innerhalb der Verzeichnisstruktur der MLDesigner-Bibliothek *Architecture Block Set* zusammengefasst (siehe Anhang B). Dies hat den Vorteil, dass die Erweiterung schnell importiert und als Plug-In eingebunden werden kann. Die Begriffswahl basiert dabei auf der Tatsache, dass Blockflussdiagramme für Entwurf und Analyse von Architekturkomponenten verwendet werden.

Als Ausgangspunkt zur Durchführung der Implementation dient die in Abschnitt 4.5 vorgenommene Formalisierung. Hieraus lassen sich zunächst die zur Umsetzung einer Basisbibliothek und Architekturbibliothek benötigten MLDesigner-Modellelemente ableiten. Für die Anwendung beider Bibliotheken, sprich den Entwurf von Architekturmodellen, ist ein zusätzliches kontrollierendes Element zur gemeinsamen Konfiguration, Verifikation und Analyse der Modellelemente erforderlich. Dies wird durch die Implementation eines neuen MLDesigner-Targets, dem sogenannten ESL-Target, erreicht (siehe auch Abschnitt 4.4). Des weiteren ist ein Architektur-Generator notwendig, durch welchen die im Rahmen des Entwurfsprozesses definierte Architekturiteration ermöglicht wird. Dabei handelt es sich um ein MLDesigner-externes Programm zur automatisierten Erstellung unterschiedlicher Architekturmodelle. Wie bereits am Ende von Abschnitt 4.5 erwähnt, definieren die vier Komponenten in Kombination das Architecture Block Set, wobei jede eine spezifische Funktionalität zur Umsetzung des Gesamtkonzepts erfüllt. Folgen-

de Übersicht listet die Komponenten auf:

- Die *Basisbibliothek* stellt standardisierte Modellkomponenten für den Entwurf von Architekturkomponenten und deren Anbindung an die gemeinsame Analyseplattform zur Verfügung.
- Alle standardisiert entworfenen Ausführungs- und Kommunikationskomponenten werden in der *Architekturbibliothek* zusammengefasst.
- Das *ESL-Target* wird zur Konfiguration, Verifikation und Analyse realistischer Gesamtsystemmodelle verwendet. Schwerpunkt der Analyse liegt auf der Bewertung architektureller Ressourcenauslastungen.
- Der *Architektur-Generator* wird zur Iteration über Architekturmodelle verwendet, wobei als Ausgangspunkt speziell aufbereitete Funktionsmodelle dienen.

Komponente	**MLDesigner-Erweiterungsart**
Basisbibliothek	Modellbibliothek
ESL-Target	Target
Architekturbibliothek	Modellbibliothek
Architektur-Generator	externer Programmaufruf

Tabelle 4.1: Komponenten der Anwendungsumgebung

Zur Umsetzung der Komponenten werden die in 4.2.6 definierten Erweiterungsmöglichkeiten von MLDesigner verwendet. Tabelle 4.1 zählt die Komponenten auf und setzt sie mit der jeweils verwendeten Erweiterungsart in Beziehung. Detaillierte Erläuterungen bezüglich der Umsetzung und Verwendung folgen in den nächsten Abschnitten.

4.6.2 Basisbibliothek

Aufgabe der Basisbibliothek ist die Bereitstellung standardisierter Modellelemente, welche beim Entwurf kompatibler, austauschbarer Architekturkomponenten zu verwenden sind. Dabei determinieren die Assoziationen zwischen den Modellelementen die architekturelle Struktur der ausführbaren Gesamtsystemspezifikation. Wie später noch gezeigt wird, spielt die standardisierte Erfassung eine zentrale Rolle bei der Verifikation und Generierung von Architekturmodellen. Tabelle 4.2 zählt alle Modellelemente der Basisbibliothek in

Bezug zum jeweiligen MLDesigner-Modelltyp auf. Im weiteren Verlauf werden diese detailliert erläutert.

Modellelement	**MLDesigner-Modelltyp**
ExecuterInterface	Primitive
ExecuterPacket	Datenstruktur
ProtocolFunctionInterface	Primitive
FunctionPacket	Datenstruktur
ProtocolChannelInterface	Primitive
ChannelPacket	Datenstruktur
ChannelInterface	Primitive
AverageLogging	Primitive
ValueLogging	Primitive

Tabelle 4.2: Komponenten der Basisbibliothek

Das erste Modellelement ist das in Abbildung 4.10 dargestellte Primitive `ExecuterInterface`. Es repräsentiert eine standardisierte Schnittstelle für den Entwurf unterschiedlicher Ausführungseinheiten, durch welche das Scheduling zur Ausführung der partitionierten Blöcke bzw. Tasks definiert wird. Dabei unterscheidet man zwischen einer externen und internen Schnittstelle. Erstere wird durch die Multiports `DataIn` und `DataOut` definiert und erlaubt den Empfang bzw. die Versendung beliebiger Daten von den bzw. an die partitionierten Blöcke. Aufgrund der notwendigen Kompatibilität zu allen MLDesigner-Datentypen verwenden beide Ports den Datentyp `Anytype`. Die Verbindung der partitionierten Blöcke mit den genannten Ports wird zu Simulationsbeginn automatisch durch das ESL-Target durchgeführt. Eine gegebenenfalls erforderliche Datentypauflösung wird ebenfalls durch das ESL-Target unterstützt. Nähere Erläuterungen hierzu erfolgen später. Die interne Schnittstelle besteht aus den Ports `PacketIn` und `PacketOut` und empfängt bzw. sendet Datenstrukturen vom Typ `ExecuterPacket` von den bzw. an die internen Blöcke der Ausführungseinheit. Zusätzlich existieren noch die vom Primitive `RepeatStar` abgeleiteten Ports `feedbackIn` und `feedbackOut`, welche zur Anbindung an den Logging-Mechanismus des ESL-Targets benötigt werden.

Zur Laufzeit verpackt das `ExecuterInterface` alle am Multiport `DataIn` empfangenen Pakete in eine Datenstruktur vom Typ `ExecuterPacket` und versendet diese über den Port `PacketOut`. Die Pakete können beispielsweise durch einen weiteren Block empfangen und in eine Warteschlange aus

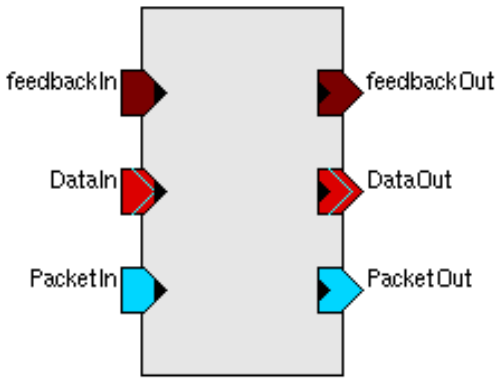

Abbildung 4.10: Primitive ExecuterInterface

abzuarbeitenden Anforderungen eingegliedert werden. Nach erfolgreicher Abarbeitung werden die Pakete wieder zum `ExecuterInterface` gesendet, dort am Port `PacketIn` empfangen und über den Multiport `DataOut` an die entsprechenden partitionierten Blöcke versendet. Daraus folgt, dass die Datenstruktur `ExecuterPacket` Adressierungsinformationen über den sendenden und empfangenden Port enthalten muss. Tabelle 4.3 zeigt die Bestandteile im Überblick. Das Feld `Data` beinhaltet einen Zeiger auf die empfangenen Daten, die Felder `SenderPort` und `ReceiverPort` jeweils einen Zeiger auf den sendenden bzw. empfangenden Port der partitionierten Blöcke. Grundlage

Feldname	**MLDesigner-Datentyp**
Data	Root.Pointer
SenderPort	Root.Pointer
ReceiverPort	Root.Pointer

Tabelle 4.3: Datenstruktur ExecuterPacket

der eindeutigen Adressierung bilden die Multiports `DataIn` und `DataOut`. Multiports werden zur Simulationszeit entsprechend der Anzahl verbundener Ports zu einzelnen Ports aufgelöst, was eine separate Verwendung zur Datenübertragung ermöglicht. Durch das `ExecuterPacket` wird ein einheitlicher Datenfluss von bzw. zur internen Schnittstelle des `ExecuterInterface` sichergestellt.

Ein weiteres Modellelement der Basisbibliothek ist das in Abbildung 4.11 dargestellte Primitive `ProtocolFunctionInterface`. Es definiert eine standardi-

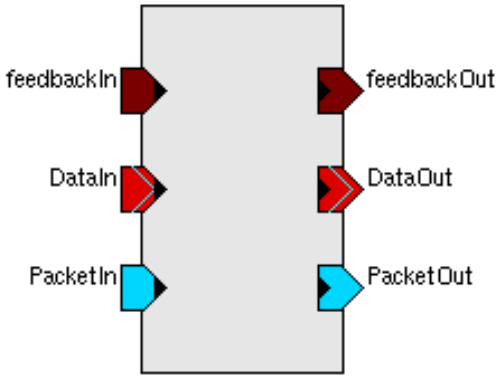

Abbildung 4.11: Primitive ProtocolFunctionInterface

sierte Schnittstelle für den Entwurf von Protokollstapeln in Richtung der Anwendungsschicht, sprich den partitionierten Blöcken. Auch hier wird ähnlich dem `ExecuterInterface` zwischen einer externen und internen Schnittstelle unterschieden. Die externe Schnittstelle wird durch die Multiports `DataIn` und `DataOut` beschrieben und ermöglicht den Empfang bzw. die Versendung beliebiger Daten von den bzw. an die partitionierten Blöcke. Zur Einhaltung der Datentypkompatibilität bezüglich aller verbundenen Blöcke, kommt der universelle Datentyp `Anytype` zum Einsatz. Hieraus folgt die Notwendigkeit zur Datentypauflösung durch das ESL-Target. Die interne Schnittstelle wird durch die Ports `PacketIn` und `PacketOut` definiert und erlaubt den Empfang bzw. die Versendung von Datenstrukturen vom Typ `FunctionPacket` von den bzw. an die internen Blöcke des Protokollstapels. Des weiteren existieren die abgeleiteten Ports `feedbackIn` und `feedbackOut`, durch welche die Anbindung an den Logging-Mechanismus des ESL-Targets realisiert wird.

Zur Konfiguration stellt das `ProtocolFunctionInterface` drei Parameter zur Verfügung. Tabelle 4.4 listet diese in Bezug zum gewählten MLDesigner-Datentyp auf. Durch den Parameter `Delay` kann die Verzögerung beim Versand von Datenstrukturen über den Port `PacketOut` festgelegt werden. Der Parameter `FixedDataSize` erlaubt die Definition einer fixen Bit-Größe für alle ankommenden Daten. Anhand des Parameters `Mapping` lässt sich der Mapping-Mechanismus des ESL-Targets aktivieren bzw. deaktivieren. Dieser kontrolliert die Zuordnung von sendenden zu empfangenden Funktionen in Form einer global bekannten Port-Liste. Hierdurch wird sichergestellt, dass die zu übertragenden Daten nur zu determinierten Empfänger-Protokollstapeln bzw. den damit verbundenen Blöcken gesendet werden. Ist

das Mapping deaktiviert, werden die Daten zu allen verbundenen Blöcken gesendet. Wie sich später noch zeigen wird, ist dieses Feature zur Umsetzung des Architektur-Generators unverzichtbar.

Parametername	**MLDesigner-Datentyp**
Delay	Float
FixedDataSize	Integer
Mapping	Enum

Tabelle 4.4: Parameter des ProtocolFunctionInterface

Kommt es zur Simulation, werden die am Multiport `DataIn` eintreffenden Daten in Datenstrukturen vom Typ `FunctionPacket` verpackt und über den Port `PacketOut` an die internen Blöcke versendet. Umgekehrt werden die am Port `PacketIn` ankommende Datenstrukturen entpackt und die Daten über den Multiport `DataOut` nach außen geleitet. Durch das `FunctionPacket` wird somit ein vereinheitlichter Datenfluss von bzw. zur internen Schnittstelle des `ProtocolFunctionInterface` gewährleistet. Tabelle 4.5 zählt die Bestandteile des `FunctionPacket` auf. Das Feld `Data` beinhaltet einen Zeiger auf die empfangenen bzw. zu übertragenen Daten, das Feld `Size` enthält die Größe der Daten in Bit, das Feld `ArrivalTime` die Zeit des DE-Schedulers beim Eintreffen der Daten und das Feld `ReceiverPort` Adressierungsinformationen in Form eines Zeigers auf den empfangenden Port. Grundlage zur Ermittlung des Port-Zeigers und damit zur eindeutigen Adressierung bildet die Multiport-Semantik von `DataIn` und `DataOut`. Ausgehend von der internen Schnittstelle werden die eigentlichen protokollbeschreibenden Blöcke angesteuert. Dabei besitzt jedes Protokoll ebenfalls eine standardisierte Schnittstelle, welche ähnlich dem `ProtocolFunctionInterface` aus den Ports `DataIn`, `DataOut`, `PacketIn` und `PacketOut` besteht und sich an der Pack-Entpack-Semantik des ISO/OSI Referenzmodells aus Abschnitt 4.3.3 orientiert. Hierdurch wird zusätzlich zur Austauschbarkeit von Protokollstapeln, ein vereinfachter Austausch einzelner Protokolle bzw. ISO/OSI Schichten ermöglicht. Voraussetzung ist die Kompatibilität des Daten- und Kontrollfluss zwischen den kommunizierenden Protokollen. Anzumerken ist, dass die Einhaltung dieser Schnittstelle beim Entwurf einzelner Protokolle empfohlen wird, jedoch nicht zwangsläufig notwendig ist. Eine Überprüfung durch den später erläuterten Verifikations-Mechanismus des ESL-Targets wird demzufolge nicht durchgeführt.

Im folgenden wird das in Abbildung 4.12 veranschaulichte Modellelement `ProtocolChannelInterface` erläutert. Dieses ist analog zum Modellelement

Feldname	MLDesigner-Datentyp
Data	Root.Pointer
Size	Integer
ArrivalTime	Float
ReceiverPort	Root.Pointer

Tabelle 4.5: Datenstruktur FunctionPacket

`ProtocolFunctionInterface` ein grundlegender Bestandteil jedes Protokollstapels und definiert dessen standardisierte Schnittstelle in Richtung der Bitübertragungsschicht, also dem verbundenen Kanal. Auch hier wird zwischen einer internen und externen Schnittstelle unterschieden. Erstere besteht aus den Ports `DataIn` und `DataOut` und erlaubt den Empfang bzw. die Versendung beliebiger Datenstrukturen von den bzw. an die verbundenen protokollbeschreibenden Blöcke. Letztere wird durch die Ports `PacketIn` und `PacketOut` definiert und gewährleistet in Verbindung mit der Datenstruktur `ChannelPacket` einen vereinheitlichten Datenfluss zum verbundenen Kanal. Zusätzlich existieren die vom Primitive `RepeatStar` abgeleiteten Ports `feedbackIn` und `feedbackOut`, welche zur Anbindung an den Logging-Mechanismus des ESL-Targets notwendig sind.

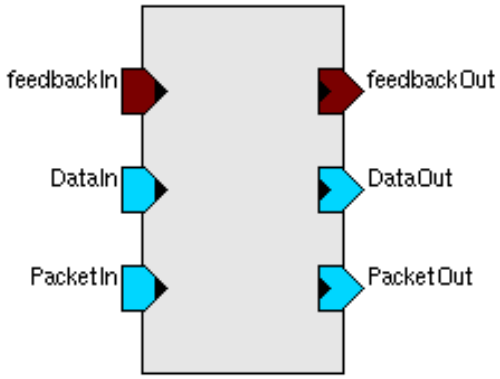

Abbildung 4.12: Primitive ProtocolChannelInterface

Kommt es zur Systemausführung, verpackt das `ProtocolChannelInterface` die am Port `DataIn` empfangenen Daten in ein `ChannelPacket` und versendet diese im Anschluss über den Port `PacketOut` an den verbundenen Kanal. Umgekehrt werden die am Port `PacketIn` empfangenen Datenstrukturen entpackt und die enthaltenen Daten über den Port `DataOut` an die internen

protokollbeschreibenden Blöcke gesendet. Tabelle 4.6 listet die Bestandteile des `ChannelPacket` auf. Das Feld `Data` beinhaltet die zu übertragenden

Feldname	**MLDesigner-Datentyp**
Data	Root
Delay	Float
SenderBlock	Root.Pointer
DispatchMode	Root.ENUM.DispatchMode

Tabelle 4.6: Datenstruktur ChannelPacket

Daten, das Feld `Delay` die durch das physikalische Übertragungsmedium entstandene Verzögerung, das Feld `SenderBlock` einen Zeiger auf den sendenden Block, sprich das This-Objekt des `ProtocolChannelInterface`, und das Feld `DispatchMode` einen Wert bezüglich der Interpretation der ermittelten Verzögerungszeit. Dabei determiniert das `ProtocolChannelInterface` lediglich die Felder `Data` und `SenderBlock`. Die restlichen Felder werden während des Kanaldurchlaufs entsprechend dem gewählten Übertragungsmedium ergänzt. Auch die Verarbeitung erfolgt hauptsächlich innerhalb des Kanals. Das `ChannelPacket` bildet somit die Basis zur Verbindung der Modellelemente `ProtocolChannelInterface` und `ChannelInterface`. Gleichzeitig wird es innerhalb des modellierten Kanals zur Sicherstellung eines vereinheitlichten Datenflusses verwendet.

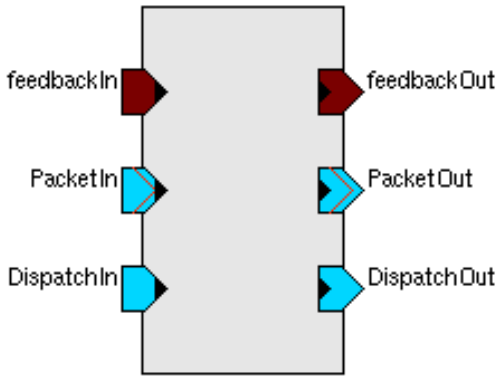

Abbildung 4.13: Primitive ChannelInterface

Im weiteren Verlauf wird das in Abbildung 4.13 dargestellte Modellelement `ChannelInterface` erläutert. Dieses definiert eine standardisierte Schnittstelle für den Entwurf von Kanälen und besteht aus einer externen und internen

Schnittstelle. Erstere wird durch die Multiports `PacketIn` und `PacketOut` beschrieben und ermöglicht den Empfang bzw. die Versendung von Datenstrukturen des Typs `ChannelPacket` von den bzw. an die verbundenen Protokollstapel. Letztere besteht aus den Ports `DispatchIn` und `DispatchOut` und ermöglicht ebenfalls den Empfang bzw. die Versendung von Datenstrukturen des Typs `ChannelPacket`, jedoch in Richtung der internen kanalbeschreibenden Blöcke. Des weiteren existieren auch hier die zur Anbindung an den Logging-Mechanismus des ESL-Targets notwendigen abgeleiteten Ports `feedbackIn` und `feedbackOut`.

Parametername	**MLDesigner-Datentyp**
MinParticipants	Integer
MaxParticipants	Integer
SenderEcho	Enum

Tabelle 4.7: Parameter des ChannelInterface

Das `ChannelInterface` besitzt drei Parameter zur Anpassung an das aktuell zu entwickelnde Kanalmodell. Tabelle 4.7 listet diese in Bezug zum verwendeten MLDesigner-Datentyp auf. Anhand der Parameter `MinParticipants` und `MaxParticipants` kann die zulässige Anzahl mit dem Kanal verbundener Protokollstapel konfiguriert werden, was bedeutet, dass die unterstützte Netzwerktopologie festgelegt wird. Beispielsweise müssen bei bidirektionalen Verbindungen, wie Ethernet, beide Parameter auf den Wert 2 gesetzt werden. Soll eine beliebige Anzahl an Verbindungen unterstützt werden, sind beide Parameter auf den Wert -1 zu setzen. Des weiteren erlaubt der Parameter `SenderEcho` die Aktivierung bzw. Deaktivierung von Sendeechos auf dem Kanal. Diese treten auf, wenn ein Endsystem bzw. Protokollstapel Daten auf ein physikalisches Übertragungsmedium bzw. den Kanal sendet und diese unmittelbar danach als Echo empfängt. Echos treten in allen Shared-Medium-Netzen auf. Ein Beispiel hierfür ist das Ethernet 10Base2 (Koaxial). Dort werden zum Senden und Empfangen die gleichen Adern verwendet, mit anderen Worten das gleiche Übertragungsmedium. Im Gegensatz hierzu werden beim Ethernet 10BaseT (Twisted-Pair) separate Adern verwendet, weswegen ein Echo ausgeschlossen ist. Beim Entwurf von Kanälen kann dieser Sachverhalt zur Beschleunigung der Simulationsgeschwindigkeit eingesetzt werden, da bei Deaktivierung der Sendeechos weniger Ereignisse auftreten und demzufolge der DE-Scheduler weniger verarbeiten muss. Anzumerken ist, dass der Parameter `SenderEcho` standardmäßig aktiviert ist.

Während der Simulation versenden die mit dem Kanal, genauer gesagt dem

`ChannelInterface`, verbundenen Protokollstapel Datenstrukturen vom Typ `ChannelPacket`. Diese werden am Port `PacketIn` empfangen und unmittelbar über den Port `DispatchOut` an die internen kanalbeschreibenden Blöcke weitergeleitet. Dabei kommt außerhalb und innerhalb des Kanals die Datenstruktur `ChannelPacket` zum Einsatz. Grund für die zweimalige Verwendung ist die Tatsache, dass der Kanal die unterste Schicht des ISO/OSI Referenzmodells beschreibt und demzufolge keine weitere Fragmentierung der Daten erforderlich ist. Des weiteren wird eine Vereinheitlichung des Datenflusses erreicht, wodurch die Zielstellung, Architekturkomponenten standardisiert zu entwerfen, zusätzlich unterstützt wird. Aufgabe der kanalbeschreibenden Blöcke ist es nun, die empfangenen Datenstrukturen zur Berechnung der Verzögerung auszuwerten und die Ergebnisse in das Feld `Delay` zu schreiben. Im Anschluss werden die Datenstrukturen an den Port `DispatchIn` des `ChannelInterface` gesendet, dort verzögert und über den Port `PacketOut` an die verbundenen Protokollstapel gesendet. Ist der Parameter `SenderEcho` deaktiviert, erfolgt keine Versendung an den ausgehenden Protokollstapel, dessen Zeiger über das Feld `SenderBlock` ermittelt werden kann. Basis der Adressierung bildet hierbei wiederum die bereits erläuterte Multiport-Semantik. Eine weitere Besonderheit beim Kanalentwurf stellt das in der Datenstruktur `ChannelPacket` enthaltene Feld `DispatchMode` dar. Dabei handelt es sich um eine Enumeration, welche die Werte `SendDelayed`, `SendUndelayed` und `DelayOnly` annehmen kann. Der `DispatchMode` erlaubt die Konfiguration des Verzögerungs- und Sendeverhalten des `ChannelInterface` und wurde aus Kompatibilitätsgründen zu bestehenden Kanalmodellen eingeführt. Beispielsweise wird in einigen Kanalmodellen nicht nur die Verzögerung berechnet, sondern unmittelbar durch die internen Blöcke realisiert. In diesem Fall darf keine Verzögerung durch das `ChannelInterface` erfolgen.

Damit ist die Beschreibung der grundlegenden Modellelemente der Basisbibliothek abgeschlossen. Im weiteren Verlauf wird kurz auf die ebenfalls zur Basisbibliothek gehörenden Logging-Modellelemente eingegangen. Durch diese können beliebige, für die Performance-Analyse relevante, Attribute an den Logging-Mechanismus des ESL-Targets angebunden werden. Alternativ ist auch eine direkte Anbindung über die Programmierschnittstelle möglich. Aufgabe des Logging-Mechanismus ist die Sicherstellung einer vereinheitlichten und synchron ablaufenden Performance-Analyse, um unterschiedliche Architekturalternativen miteinander vergleichen zu können. Aktuell existieren zwei Logging-Modellelemente, obgleich innerhalb der Programmierschnittstelle weitere definiert sind. Dabei handelt es sich um die Primitives `ValueLogging` und `AverageLogging`. Beide besitzen einen Parameter zu Festlegung eines eindeutigen Attribut-Bezeichners, welcher zusammen mit dem anmel-

denden Block zur Identifizierung durch das ESL-Target verwendet wird. Das `ValueLogging` erlaubt die Überwachung eines beliebigen Attributes während der Ausführung des Systems. Durch das `AverageLogging` wird automatisch der Durchschnitt aller erfassten Werte eines Attributes über die gesamte Simulationszeit ermittelt, wobei im Vorab ein bestimmtes Bezugsintervall festzulegen ist. Nähere Erläuterungen über die Funktionsweise des Logging-Mechanismus erfolgen später.

4.6.3 Architekturbibliothek

Die Architekturbibliothek besteht aus einer beliebig erweiterbaren Menge standardisiert entworfener Architekturkomponenten. Dabei wird zwischen Ausführungseinheiten, Protokollstapeln und Kanälen unterschieden. Beispiele für den Entwurf sind unter anderem in [Eck 2007] und [Lohse 2007] zu finden. Ziel ist es, einen flexiblen Austausch und die Wiederverwendung durch verschiedene Entwurfsalternativen sicherzustellen. Da zur Einhaltung des Standards determinierte Schnittstellen notwendig sind, basiert die Realisierung auf den im Vorab definierten Modellelementen der Basisbibliothek. Anzumerken ist, dass Partitionen aufgrund der integrierten anwendungsspezifischen Funktionen nicht zur Architekturbibliothek gehören. Im weiteren Verlauf wird die konstitutive Struktur der genannten Architekturkomponenten ausführlich erläutert.

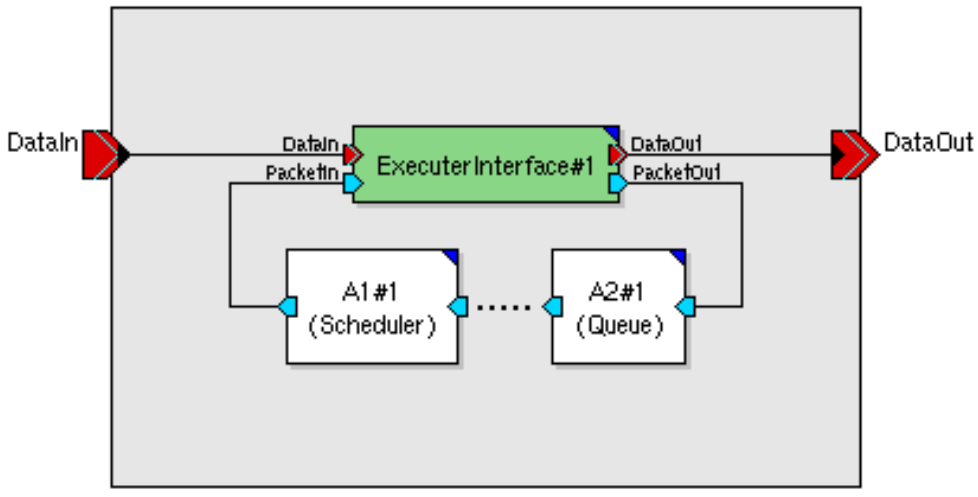

Abbildung 4.14: Ausführungseinheit

Ausführungseinheiten ermöglichen den Entwurf unterschiedlicher Betriebssysteme zur ressourcenabhängigen Ausführung von Funktionen bzw. Tasks. Abbildung 4.14 stellt den Aufbau beispielhaft dar. Die Schnittstelle mit der Umgebung, auch externe Schnittstelle genannt, wird durch die Multiports `DataIn` und `DataOut` definiert. Beide sind mit den semantisch äquivalenten Multiports des ebenfalls zur Ausführungseinheit gehörenden Blocks `ExecuterInterface` verbunden. Dieser besitzt zusätzlich die Ports `PacketIn` und `PacketOut`, durch welche eine einheitliche interne Schnittstelle definiert wird. Zur Beschreibung der eigentlichen Architektureigenschaften kommen beliebige nutzerspezifische Blöcke zum Einsatz. Durch Queue-Blöcke können abzuarbeitende Tasks zwischengespeichert werden, Ressource-Blöcke ermöglichen die Belegung bzw. Freigabe von Ressourcen, Scheduler-Blöcke arbeiten die Tasks entsprechend der definierten Strategie ab. Dabei zeichnen sich Ausführungseinheiten untereinander durch ähnliche Struktur- und Verhaltensmuster aus. Folgende Übersicht veranschaulicht den grundsätzlichen Ablauf einer Task-Ausführung:

- Eintreffen neuer Daten am `ExecuterInterface` und Versendung eines `ExecuterPacket` bzw. Tasks über die interne Schnittstelle
- Konfiguration der durch den Task benötigten Ressourcen
- Einreihung des Tasks in die Warteschlange
- Ermittlung und Aktivierung des abzuarbeitenden Tasks, gegebenenfalls Suspendierung des aktuellen Tasks, Weiterleitung des entsprechenden `ExecuterPacket`
- Allokation der geforderten Ressourcen
- Ermittlung und Zuordnung der benötigten Ausführungszeit, Weiterleitung des `ExecuterPacket`
- Freigabe der verwendeten Ressourcen
- Eintreffen des `ExecuterPacket` am `ExecuterInterface` und verzögerte Versendung der Daten über die externe Schnittstelle

Als Anwendungsbeispiel sind unter anderem Betriebssysteme zu nennen, die nach dem Standard für offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK) [Zimmermann u. Schmidgall 2007] entworfen sind. Dabei ist zu beachten, dass jede Ausführungseinheit anhand von real existierenden Systemen validiert werden muss.

Durch Protokollstapel können die zur Datenübertragung notwendigen Protokolle und deren Ressourcen kombiniert entworfen werden. Grundlage bilden die Schichten des ISO/OSI Referenzmodells sowie die bereits erläuterte Pack-Entpack-Semantik. Dabei gehört die Bitübertragungsschicht nicht zum Protokollstapel, da diese dem Übertragungsmedium entspricht und separat durch das Modellelement Kanal repräsentiert wird. Abbildung 4.15 stellt den Aufbau eines Protokollstapels beispielhaft dar. Die Schnittstelle in Richtung

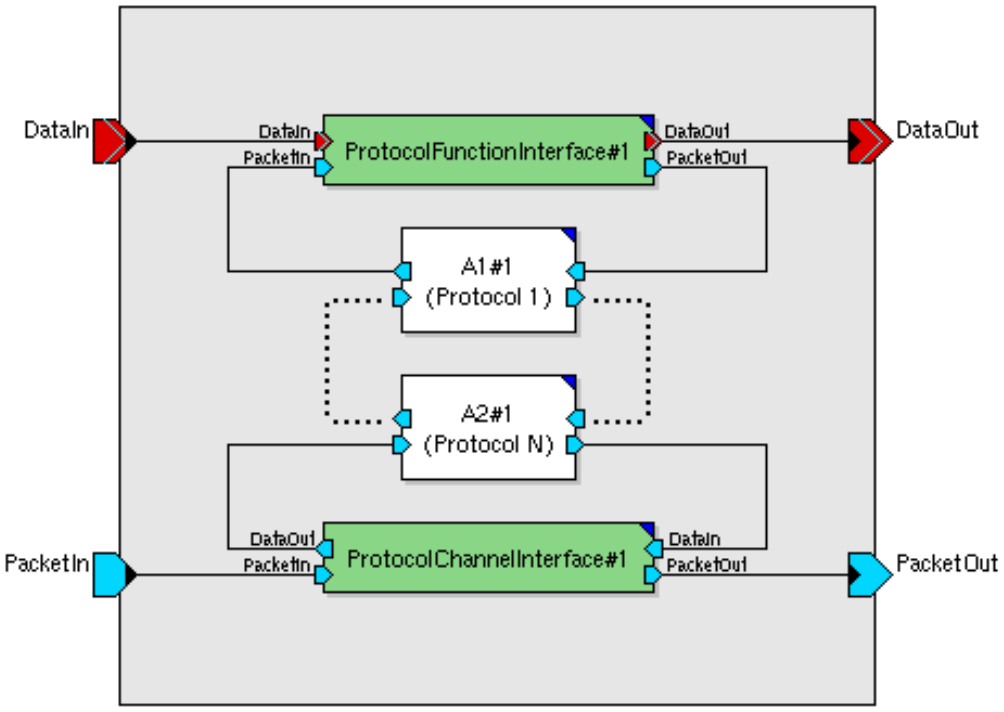

Abbildung 4.15: Protokollstapel

der kommunizierenden Funktionen und in Richtung des Kanals, sprich die externe Schnittstelle, wird durch die Multiports `DataIn` und `DataOut` bzw. die Ports `PacketIn` und `PacketOut` definiert. Erstere sind mit den gleichnamigen Multiports des `ProtocolFunctionInterface`, letztere mit den Ports des `ProtocolChannelInterface` verbunden. Beide Blöcke gehören ebenfalls zum Protokollstapel und sind simultan an der Definition der internen Schnittstelle beteiligt. Diese besteht aus den zum `ProtocolFunctionInterface` gehörenden Ports `PacketIn` und `PacketOut`, sowie den zum `ProtocolChannelInterface` gehörenden Ports `DataIn` und `DataOut`. Die Beschreibung der eigentlichen Architektureigenschaften erfolgt durch nutzerspezifische Blöcke. Auch

hier differenziert man, analog zu den oben erläuterten Ausführungseinheiten, zwischen Queue-, Ressource- und Scheduler-Blöcken, jedoch sind diese in unterschiedliche Protokoll-Module integriert. Hervorzuheben ist, dass sämtliche Protokollstapel eine vergleichbare Struktur- und Verhaltensbeschreibung besitzen. Folgende Übersicht schildert den charakteristischen Ablauf eines Sendevorgangs:

- Eintreffen neuer Daten am `ProtocolFunctionInterface` und Versendung eines `FunctionPacket` bzw. einer Transmissionsanfrage über die interne Schnittstelle
- Gegebenenfalls Durchführung protokollspezifischer Initialisierungen unter Verwendung von Datenstrukturen
- Scheduling der Datenversendung inklusive Ressourcenverwaltung und Fragmentierung, Verpacken in protokollspezifische Datenstrukturen sowie verzögerte Versendung zum nächsten Protokoll bzw. letztlich zum `ProtocolChannelInterface`
- Eintreffen der protokollspezifischen Datenstrukturen am `ProtocolChannelInterface` und Versendung als `ChannelPacket` über die externe Schnittstelle

Beim Empfang von Daten wird der Ablaufplan entsprechend reversiv durchlaufen, wobei in diesem Fall das `ProtocolChannelInterface` bzw. das jeweilig empfangene `ChannelPacket` als Ausgangspunkt dient. Beispiele zur Anwendung sind unter anderem Übertragungsprotokolle wie Ethernet, das auf Funk basierende Zigbee, der Universal Serial Bus (USB), das im SoC-Bereich verwendete Wishbone und die im Standard Electronic Industries Alliance 232 (EIA232) definierte serielle Schnittstelle. Um entworfene Protokollstapel sinnvoll einsetzen zu können, müssen diese im Vorab anhand von real existierenden Systemen validiert werden.

Die Architekturkomponente Kanal erlaubt den ressourcenabhängigen Entwurf physikalischer Übertragungsmedien. Als Basis dient die Bitübertragungsschicht des ISO/OSI Referenzmodells. Dabei ist anzumerken, dass prinzipiell auch die darüberliegenden Schichten bzw. Protokollspezifikationen innerhalb eines Kanals erfasst werden können. Dies ist jedoch zu vermeiden, da das im Rahmen der vorliegenden Arbeit entwickelte Konzept, explizit zwischen Protokoll und Medium unterscheidet. Abbildung 4.16 zeigt die Bestandteile eines Kanals am Beispiel. Die Schnittstelle mit der Umgebung bzw. externe Schnittstelle wird durch die Multiports `PacketIn` und `PacketOut` definiert.

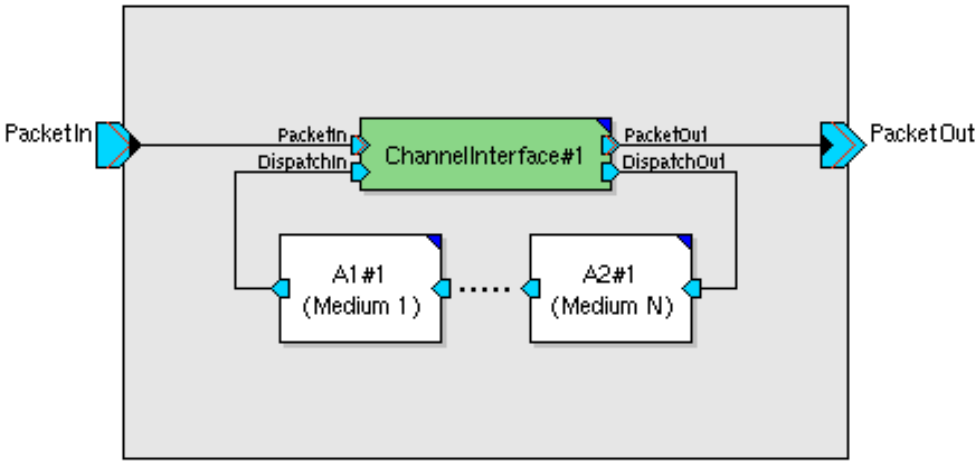

Abbildung 4.16: Kanal

Beide sind mit den gleichnamigen Multiports des Blocks `ChannelInterface` verbunden. Dieser gehört ebenfalls zum Kanal und definiert gleichzeitig die aus den Ports `DispatchIn` und `DispatchOut` bestehende interne Schnittstelle. Zur Beschreibung der Eigenschaften des Übertragungsmediums werden nutzerspezifische Blöcke verwendet. Dabei handelt es sich hauptsächlich um Ressource-Blöcke zur Ermittlung der Kanalverzögerung. Alle Kanäle besitzen eine vergleichbare Struktur- und Verhaltensbeschreibung, woraus sich ein grundlegender Ablaufplan für einen Kanalzugriff ableiten lässt. Folgende Übersicht veranschaulicht diesen:

- Eintreffen eines `ChannelPacket` am `ChannelInterface` und Versendung über die interne Schnittstelle
- Ermittlung der Verzögerung anhand der Datengröße und der verfügbaren Kanalressourcen
- Eintreffen des `ChannelPacket` am `ChannelInterface` und verzögerte Versendung über die externe Schnittstelle

Als Beispiele für den Entwurf von Kanälen sind die gleichen Übertragungsprotokolle wie bei den oben erläuterten Protokollstapeln zu nennen. Auch hier ist eine Validierung anhand realer Systeme unumgänglich, um eine sinnvolle Anwendung für Performance-Analysen zu gewährleisten.

4.6.4 ESL-Target

Das ESL-Target ist die zentrale Verwaltungseinheit aller manuell oder automatisiert erstellten architekturellen Gesamtsystemmodelle S_{Arch}. Wie bereits in 4.2 erläutert, koordinieren Targets die Systemausführung. Abbildung 4.17 stellt im oberen Teil ein Target in Beziehung zum umgebenden System und dessen Bestandteilen, also den Blöcken, dar. Dabei kann jeder Block auf das Target sowie das Target auf jeden Block zugreifen. Dieser Zusammenhang wird sich im weiteren Verlauf als wichtiges Feature erweisen. Anzumerken ist, dass Targets keine grafische Notation besitzen und lediglich anhand von Systemparametern auswählbar und konfigurierbar sind.

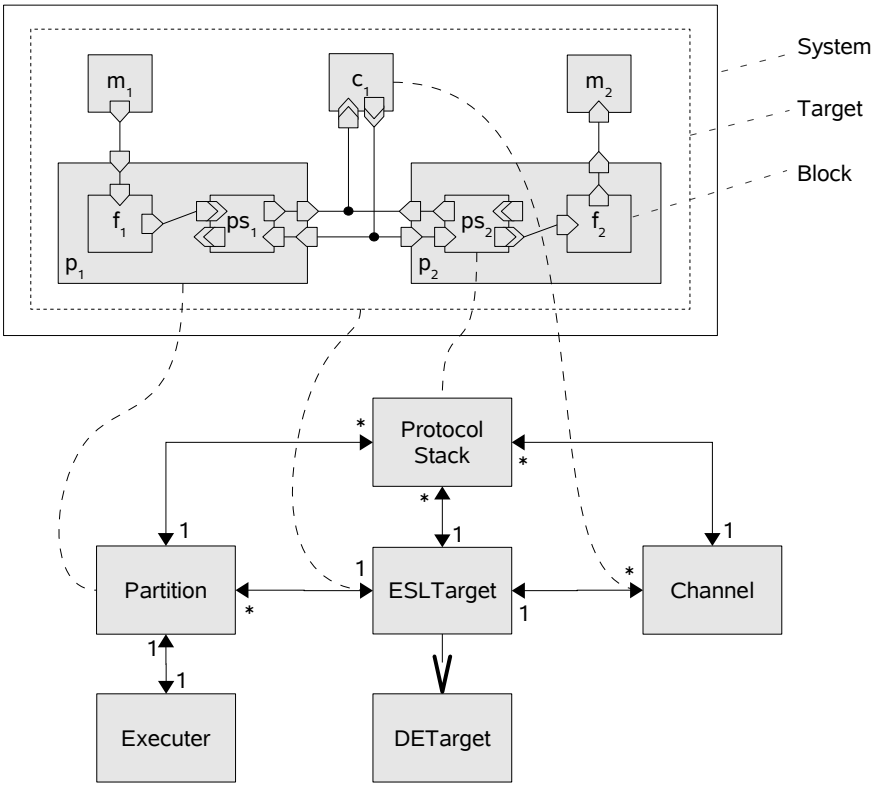

Abbildung 4.17: Beziehung zwischen ESL-Target und Modell

Durch die Einführung des ESL-Targets wird das *default-DE-Target* der DE-Domäne um architekturbezogene Informationen und Funktionen erweitert. Grund für die Auswahl des default-DE-Targets ist die Tatsache, dass hier-

durch die Ausführung aller zeitorientierter DE-Modelle und damit auch realistischer Gesamtsystemmodelle gesteuert, sowie die bereits vorhandene Funktionalität weiter verwendet werden kann. Die Umsetzung beruht auf der Klasse `ESLTarget` (siehe Anhang A), welche von der Klasse `DETarget` abgeleitet ist. Da kaum Informationen über die Vorgehensweise zur Definition neuer MLDesigner-Targets existieren, wird im folgenden kurz darauf eingegangen. Abbildung 4.18 zeigt zwei Code-Fragmente der Klasse `ESLTarget`. Im obe-

```
#include "DETarget.h"

class ESLTarget: public DETarget
{
  public:
    ESLTarget();
    ~ESLTarget() {}

    Block* makeNew() const;
}
```

```
static ESLTarget sESLTarget;
static KnownTarget sKnownTarget(sESLTarget, "ESL");

ESLTarget::ESLTarget()
{
  Target("ESL", "DEStar", "ESL␣DE␣target");
}

Block* ESLTarget::makeNew() const
{
  return new ESLTarget;
}
```

Abbildung 4.18: Auszug aus der Implementation des ESL-Targets

ren Teil wird die HPP-Datei und im unteren die CPP-Datei dargestellt. Wie leicht zu erkennen ist, wird eine Vererbung durchgeführt. Dabei ist zu beachten, dass die virtuelle Methode `makeNew` neu deklariert und definiert werden muss. Des weiteren müssen die von `DETarget` initialisierten Werte durch einen erneuten Aufruf des Konstruktors von `Target` überschrieben werden. Die Registrierung des neuen ESL-Targets am Simulations-Kernel von MLDesigner erfolgt durch Anlegen eines statischen Objekts der Klasse `ESLTarget` sowie `KnownTarget`, wobei letzteres unter Verwendung des erstgenannten konstruiert wird. Im Anschluss an die Implementation muss das ESL-Target an die Laufzeitumgebung von MLDesigner angebunden werden. Hierzu wird der

Quellcode übersetzt, in eine Shared Library verpackt und während der Ausführung von MLDesigner dynamisch geladen. Folgende Übersicht zählt die durchzuführenden Schritte auf:

- Start von MLDesigner, Import der Bibliothek `ArchitectureBlockSet` und Beenden des MLDesigner
- Öffnen der Datei `~/.mld/.mldrc`, Hinzufügen des Eintrages `~/MLD/ArchitectureBlockSet/esltarget/libesltarget.so` zwischen den `DYNAMIC LIBRARIES` Tags, Speichern und Schließen der Datei
- Wechsel in das Verzeichnis `~/MLD/ArchitectureBlockSet/esltarget`, Aufruf des Befehls `make release` zur Übersetzung des Quellcodes
- Erneuter Start von MLDesigner zur Verwendung der Bibliothek `ArchitectureBlockSet` in Verbindung mit dem ESL-Target

Im weiteren Verlauf werden die Funktionen des ESL-Targets erläutert. Dabei wird jede Funktion einer bestimmten Simulationsphase des ESL-Targets zugeordnet. Diese laufen parallel zu den normalen Simulationsphasen von MLDesigner ab und koordinieren die erweiterten architekturbezogenen Modelleigenschaften. Da hierbei zum einen auf bereits vorhandene und zum anderen auf neu eingeführte Modelleigenschaften Bezug genommen wird, kann man von einem *Modell im Modell* Ansatz sprechen. Folgende Übersicht zählt die Simulationsphasen des ESL-Targets auf, definiert diese und benennt die wichtigsten Funktionen:

- **Phase 1 – Konfigurationsphase:**
  In dieser Phase wird das System vor der Simulation manuell oder automatisiert parametrisiert. Dabei handelt es sich hauptsächlich um Mapping-Einträge und die Konfiguration der Architekturkomponenten.
- **Phase 2 – Initialisierungsphase:**
  Die Initialisierungsphase findet zu Simulationsbeginn zur Verifikation und Adaption des Systems statt. Unter anderem wird der strukturelle Aufbau und die Parametrisierung analysiert, eine Datentypauflösung durchgeführt, sowie die dynamische Einbindung von Ausführungseinheiten vollzogen.
- **Phase 3 – Ausführungsphase:**
  Innerhalb der Ausführungsphase findet die Steuerung des architekturellen Datenflussen und die Analyse bzw. Validierung des Systems statt.

Hierzu werden die Performance-Werte aller Architekturkomponenten synchronisiert aufgezeichnet und visualisiert.

Bevor auf die einzelnen Phasen sowie deren Funktionen detailliert eingegangen werden kann, muss zunächst eine gemeinsame, systemweit verfügbare Schnittstelle zur Erfassung, Analyse und Verarbeitung architekturbezogener Informationen eingeführt werden. Dabei handelt es sich um ein parallel zum Systemmodell existierendes und an das ESL-Target angebundenes Klassenmodell, dessen Definition auf der Formalisierung aus Abschnitt 4.5 basiert. Das Klassenmodell ermöglicht nicht nur die Datenhaltung von Architekturinformationen, sondern gleichzeitig eine vereinfachte Navigation zwischen Architekturkomponenten. Die Anbindung erfolgt über die Blöcke bzw. Module der Architekturbibliothek und die darin verwendeten Blöcke bzw. Primitives der Basisbibliothek. Abbildung 4.17 stellt diesen Zusammenhang dar. Im oberen Teil der Abbildung werden mehrere in ein Target sowie System eingebettete Blöcke gezeigt. Einige der Blöcke repräsentieren Architekturkomponenten und besitzen deshalb Assoziationen zu den Klassen im unteren Teil der Abbildung. Grundlage hierfür ist die Tatsache, dass das Target auf alle integrierten Blöcke zugreifen kann. Im weiteren Verlauf werden die drei Phasen des ESL-Targets erläutert, wobei das Klassenmodell mehrfach zum Einsatz kommt.

Parametername	**MLDesigner-Datentyp**
ESLDebug	Enum
ESLMonitor	Enum
ESLSetup	String
(PartitionSetup)	(String)

Tabelle 4.8: Parameter des ESL-Targets

Die *Konfigurationsphase* bezieht sich auf die Festlegung der Parameter des ESL-Targets sowie die der Partitionen. Alle anderen Parameter werden durch das neu definierte Target nicht überwacht und demzufolge auch nicht ausgewertet. Tabelle 4.8 listet die Parameter im einzelnen auf. Anhand des Parameters `ESLDebug` kann der Debug-Modus zur Anzeige zusätzlicher Informationen im Logging-Fenster von MLDesigner aktiviert werden. Diese beziehen sich hauptsächlich auf die Initialisierung des Klassenmodells, die Verifikation der Architekturbeschreibung sowie die Datenverarbeitung bzw. -übertragung. Abbildung 4.19 zeigt die Ausgabe solcher Informationen am Beispiel. Der Parameter `ESLMonitor` erlaubt die Aktivierung der in Abbildung 4.22 und

4.23 dargestellten grafischen Ausgabe zur Durchführung von Performance-Analysen. Eine Besonderheit stellen die als `String` typisierten Parameter `ESLSetup` und `PartitionSetup` dar. Beide werden vom ESL-Target zeilenwei-

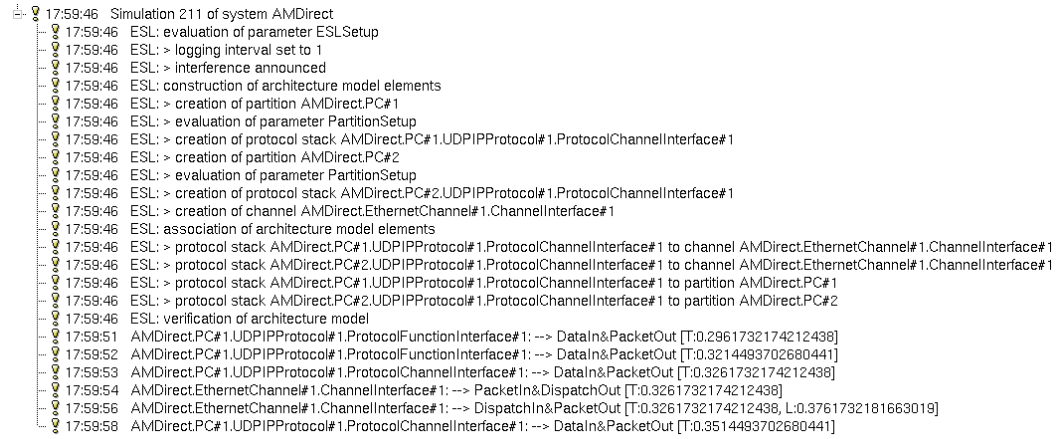

Abbildung 4.19: Debug-Ausgaben des ESL-Targets

se als reguläre Ausdrücke interpretiert und zur Konfiguration verschiedener Funktionen verwendet. Dabei ist anzumerken, dass der Parameter `PartitionSetup` eigentlich Bestandteil verschiedener Partitions-Blöcke ist, jedoch aufgrund seines exklusiven Bezugs zum ESL-Target mit in die Tabelle 4.8 aufgenommen wurde. Im folgenden werden zunächst die Interpretationsmöglichkeiten des Parameters `ESLSetup` erläutert. Dieser erlaubt unter anderem die Determinierung des Update-Intervalls für den gemeinsamen, synchronisierten Logging-Mechanismus:

$$logginginterval = value;$$

mit:

$value$ = positive reelle Zahl

Beispiel:

$$logging = 2.5;$$

Des weiteren können Mapping-Einträge zur Verbindung und Adressierung von Funktionen definiert werden. Handelt es sich um ein generiertes Modell, werden die entsprechenden Mapping-Einträge automatisch gesetzt. Bei manuell erstellten Modellen müssen diese für alle assoziierten Funktionsblöcke durch den Nutzer hinzugefügt werden (siehe auch Abbildung 4.7). Hervorzu-

heben ist, dass auch eine dynamische Ermittlung möglich ist. Hierbei werden die Port-Relationen des parallel vorhandenen funktionalen Gesamtsystemmodells S_{Func} während der Initialisierungsphase ausgewertet. Die Definition eines Mapping-Eintrags erfolgt anhand folgender Regel:

$$mapping = portfullname, portfullname;$$

mit:

$portfullname$ = voll qualifizierter Port-Name in MLDesigner

Beispiel:

$$mapping = System.BlockA\#1.PortA, System.BlockB\#1.PortA;$$

Durch Interferenzbereiche können probabilistische Störungen in einem dreidimensionalen Raum positioniert werden. Diese beeinflussen die Kommunikation zwischen Partitionen in Abhängigkeit der Position. Ursachen für Interferenzen können innerhalb des eigenen Systems oder bei der Umgebung liegen. Zur Definition wird folgende Regel verwendet:

$$interference = xpos, ypos, zpos, xstretch, ystretch, zstretch, value;$$

mit:

$xpos, ypos, zpos, xstretch, ystretch, zstretch$ = positive reelle Zahl
$value$ = positive reelle Zahl im Intervall [0, 1]

Beispiel:

$$interference = 5.2, 7, 3.4, 3, 1.1, 3, 0.35;$$

Analog zum Parameter `ESLSetup` verwendet auch der Parameter `PartitionSetup` verschiedene reguläre Ausdrücke zur Konfiguration des ESL-Targets. Dabei ist zu beachten, dass jeder Block der den Parameter besitzt eine Partition repräsentiert, welche gleichzeitig durch diesen konfiguriert wird. Unter anderem erlaubt der Parameter die Zuordnung einer Ausführungseinheit, um die partitionierten Funktionskomponenten bzw. Tasks architekturspezifisch auszuführen. Die Zuordnung erfolgt anhand folgender Regel:

$$executer = modelpath;$$

mit:

$modelpath$ = Pfad zu einem MLDesigner-Modell

Beispiel:

$$executer = \$MLD\_USER/ExecuterA/ExecuterA.mml;$$

Zudem kann jeder Partition eine dreidimensionale Position zugewiesen werden, welche die Grundlage zur Verwendung einer topologischen Modelldarstellung sowie zur Ermittlung und Akkumulation von Interferenzen zwischen kommunizierenden Partitionen darstellt. In diesem Zusammenhang kommt folgende Regel zum Einsatz:

$$position = xpos, ypos, zpos;$$

mit:

$$xpos, ypos, zpos = \text{positive reelle Zahl}$$

Beispiel:

$$position = 2, 3.4, 3;$$

Im Anschluss an die Konfigurationsphase folgt die *Initialisierungsphase*. Diese besteht ebenfalls aus mehreren Funktionen, die im weiteren Verlauf entsprechend ihrer Ausführungsreihenfolge erläutert werden. Dabei wird als erstes die blockorientierte Architekturstruktur unter Verwendung bestimmter Regeln verifiziert. Wird eine Regel nicht erfüllt, kommt es zum Abbruch der Simulation. Ausgangspunkt zur Definition der Regeln bildet die Formalisierung aus Abschnitt 4.5. Die wichtigsten Regeln lauten:

- Jeder Partition p muss eine Ausführungseinheit e zugeordnet werden
- Jeder Protokollstapel ps muss Bestandteil einer Partition p sein
- Jede Partition p muss mindestens einen Protokollstapel ps besitzen
- Jeder Kanal c muss mindestens zwei Protokollstapel ps verbinden

Zur Umsetzung kommt das oben beschriebene Klassenmodell zum Einsatz, welches die notwendigen strukturellen Informationen implizit enthält. Anzumerken ist, dass sich die Architekturstruktur in der Baumstruktur des Monitors aus Abbildung 4.22 widerspiegelt.

Daraufhin wird die Verifikation spezifischer nichtstruktureller Architektureigenschaften für alle in Relation stehende Architekturkomponenten durchgeführt. Hierbei müssen die zu überprüfenden Eigenschaften bzw. Parameter

bereits während des Entwurfs der Architekturkomponenten festgelegt werden. Die Umsetzung beruht wiederum auf dem oben erläuterten Klassenmodell. Diesem werden alle notwendigen Eigenschaften zugeordnet, um anschließend eine Überprüfung anhand beliebig definierbarer Regeln durchzuführen. Dabei erweist sich die direkte und beschleunigte Navigation zwischen den Instanzen des Klassenmodells als vorteilhaft. Während der architekturspezifischen Verifikation werden hauptsächlich Eigenschaften zwischen Kanälen und Protokollstapeln, sowie innerhalb von Protokollstapeln überprüft. Beispielsweise kann ein Ethernet-Kanal mit Unshielded Twisted Pair (UTP) 5 und ein Protokollstapel mit dem Ethernet-Standard 10GBase-T konfiguriert werden, welche jedoch nicht kompatibel sind. Des weiteren kann ein Protokollstapel gleichzeitig mit 10GBase-T und der Betriebsart Halb-Duplex konfiguriert werden, welche ebenfalls nicht kompatibel sind. Werden solche Architekturkomponenten dennoch miteinander verbunden, kommt es während der Verifikation zu einer entsprechenden Fehlermeldung und zum Abbruch der Simulation.

Im nächsten Schritt wird in alle Partitionen die jeweils zugeordnete Ausführungseinheit automatisiert eingebunden. Ziel ist es, eine architekturspezifische Funktionsausführung durch die Verzögerung der funktionalen Block-Kommunikation zu emulieren. Da die Einbindung auf einer temporären Abänderung des Modells zur Simulationszeit beruht, wird der Vorgang auch als dynamische Integration bezeichnet. Grund hierfür ist die Tatsache, dass bei einer statischen Einbindung während des Entwurfs die Visualisierung der funktionalen Kommunikationsstrukturen verloren gehen würde. Zudem wäre ein Austausch von Ausführungseinheiten nur schwer durchführbar. Voraussetzung der dynamischen Integration ist die Erfassung aller Port-Relationen innerhalb der Partitionen. Diese werden unter Verwendung der auf Ports anwendbaren MLDesigner-Kernel-Funktion `far()` ermittelt und in Form von Port-Pointer-Mappings gespeichert. Anzumerken ist, dass die Semantik prinzipiell den oben erläuterten textuellen Mappings entspricht. Im Anschluss werden separat für jede Partition Instanzen der gewählten Ausführungseinheit integriert, sowie alle von Ports ausgehende Relationen durch zwei neue, mit der Ausführungseinheit verbundene, ersetzt. Folgende Überführungsfunktion verdeutlicht diesen Zusammenhang:

$$r_{old}(p_{b1}, p_{b2}) \rightarrow r_{new1}(p_{b1}, p_e), r_{new2}(p_e, p_{b2})$$

mit:

$r_{old}, r_{new1}, r_{new2} \in R$ = Relationen zwischen jeweils einem Input- und einem Output-Port

$p_{b1}, p_{b2}, p_e \in P$ = Ports von Blöcken bzw. Ausführungseinheiten

Wie weiter unten noch erläutert wird, bilden die neu eingefügten Relationen die Grundlage zur ressourcenabhängigen Funktionsausführung. Dabei ist zu beachten, dass die Blöcke bzw. Tasks zunächst berechnet werden und Daten produzieren. Anschließend werden diese durch die Ausführungseinheit empfangen und ressourcenabhängig verzögert. Grund für die nachgelagerte Verzögerung ist die Tatsache, dass nur so Quell-Blöcke, welche bekanntlich keine Input-Ports besitzen, erfasst werden können. Des weiteren wird vorausgesetzt, dass jede Funktion unabhängig berechnet werden kann und ausschließlich über Ports mit der Umgebung interagiert. Abbildung 4.20 stellt die dynamische Integration am Beispiel dar. Als Ausgangspunkt dient das Modell im oberen Teil von Abbildung 4.17. Man erkennt, dass bestimmte Relationen gelöscht bzw. hinzugefügt wurden. Unter anderem wurde die Relation $r_1(p_{f1}, p_{ps1})$ in die Relationen $r_1(p_{f1}, p_{e1})$ und $r_1(p_{e1}, p_{ps1})$ überführt. An dieser Stelle sei auch Abbildung 4.9 erwähnt, in welcher die dynamisch erzeugten Modellelemente gestrichelt dargestellt werden.

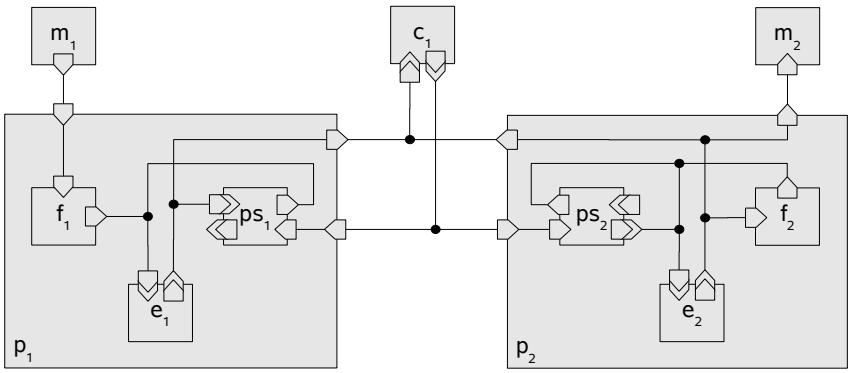

Abbildung 4.20: Dynamische Integration von Ausführungseinheiten

Die nächste Aufgabe der Initialisierungsphase ist, insofern aktiviert, die dynamische Ermittlung von Mappings zur Adressierung partitionierter Funktionen. Wie bereits oben erläutert, können Mapping-Einträge unter Verwendung des Parameters `ESLSetup` definiert werden. Alternativ ist auch die Zuordnung eines parallel vorhandenen funktionalen Gesamtsystemmodells möglich, welches als Ausgangspunkt zur dynamischen Ermittlung aller Mappings dient. In diesem Fall kann auf manuelle Mappings verzichtet werden. Dabei ist zu beachten, dass die funktionale Struktur des funktionalen Gesamtsystemmodells äquivalent zu der des aktuellen architekturellen Gesamtsystemmodells sein muss. Da alle MLDesigner-Modelle in XML gespeichert werden, kommt zur

Ermittlung der Mappings ein XSLT-Scanner zum Einsatz. Dieser iteriert über die Modellhierarchie, analysiert sämtliche in Relation stehende Ports und schreibt die Ergebnisse in eine String-Map. Anschließend werden alle Mappings, egal ob statisch oder dynamisch definiert, in Port-Pointer-Mappings transformiert, um String-Vergleiche zu vermeiden und eine performante Ausführung zu gewährleisten.

Letzte Aufgabe der Initialisierungsphase ist die Durchführung einer Datentypauflösung, ohne die eine Ausführung realistischer Gesamtsystemmodelle unmöglich ist. Diese erfolgt zusätzlich zu der im oberen Teil von Abbildung 4.21 dargestellten normalen MLDesigner-Datentypauflösung. Die dort gezeig-

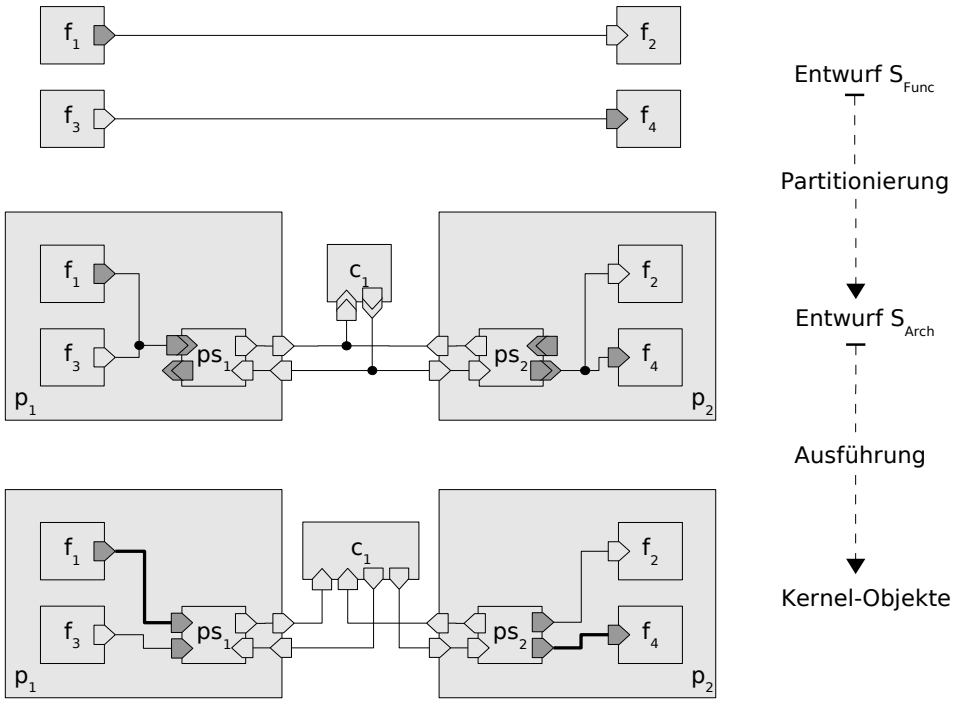

Abbildung 4.21: Datentypauflösung durch das ESL-Target

ten Relationen bestehen jeweils auf zwei Ports, wobei die farblich hervorgehobenen untypisiert sind, sprich den Datentyp `Anytype` besitzen. Ziel der Datentypauflösung ist es nun, die fehlenden Datentypen eindeutig zu bestimmen. Um dies zu erreichen, analysiert MLDesigner die Relationsnetzwerke und ordnet, insofern möglich, untypisierten Ports einen kompatiblen Daten-

typ zu. In Bezug auf das Beispiel bedeutet dies, dass der jeweilig gegenüberliegende Datentyp verwendet wird. Da es im Verlauf des Entwurfsprozesses zur Partitionierung und damit zur Integration generischer Architekturkomponenten kommt, folgt hieraus eine partielle Auftrennung der funktionalen Modellstruktur. Dieser Zusammenhang wird im zentralen, sowie inklusive aufgelösten Multiports im unteren Teil von Abbildung 4.21 dargestellt. Man erkennt, dass einige der neu entstandenen Relationen ausschließlich aus untypisierten Ports bestehen, was eine Ausführung des Modells unmöglich macht. Daraus folgt die Notwendigkeit einer weiteren, architekturübergreifenden Datentypauflösung durch das ESL-Target. Die Funktionsweise beruht auf der Tatsache, dass die im Vorab ermittelten Port-Pointer-Mappings bereits alle notwendigen Datentyp-Informationen enthalten, wodurch die der typisierten Ports auf die jeweils zugeordneten untypisierten Ports übertragen werden können. Zur Umsetzung kommt die auf Ports anwendbare MLDesigner-Kernel-Funktion `inheritTypeFrom()` zum Einsatz. Beispielsweise wird dem untypisierten Port von Funktion f_1 der Datentyp des Ports von Funktion f_2 zugewiesen. Anzumerken ist, dass die ESL-Target-Datentypauflösung auch in Bezug auf die Relationen der dynamisch integrierten Ausführungseinheiten verwendet wird.

Damit ist die Initialisierungsphase abgeschlossen und es folgt die ebenfalls aus mehreren Funktionen bestehende *Ausführungsphase*. Wichtigste Funktion ist die automatische Steuerung des architekturellen Datenflusses innerhalb sowie zwischen den Partitionen. Konkret bedeutet dies, dass jedes Datenpaket entsprechend seines Sender-Ports und den vorhandenen Adressierungsinformationen, alias Port-Pointer-Mappings, durch das Modell geleitet wird, um letztlich beim gewünschten Empfänger-Port anzukommen. Die Realisierung erfolgt durch die standardisierten Blöcke der Basisbibliothek und beruht auf Pointer-Vergleichen.

Des weiteren findet während der Ausführungsphase eine synchronisierte Bemessung und Erfassung von Performance-Werten in Form von Durchsatzraten, Ressourcenauslastungen und Verzögerungen statt. Ziel ist es, einzelne Architekturkomponenten sowie komplette Architekturmodelle analysieren, vergleichen und validieren zu können. Alle Performance-Werte werden durch sogenannte Logging-Einträge umgesetzt und automatisch am Logging-Mechanismus des ESL-Targets angemeldet. Dabei wird zwischen fest vorgegebenen und architekturspezifischen Logging-Einträgen unterschieden. Erstere sind aufgrund der standardisierten Schnittstellen Bestandteil jeder Architekturkomponente, letztere können durch die bereits erläuterten Logging-Modellelemente oder eine direkte Quellcode-Anbindung beliebig hinzugefügt

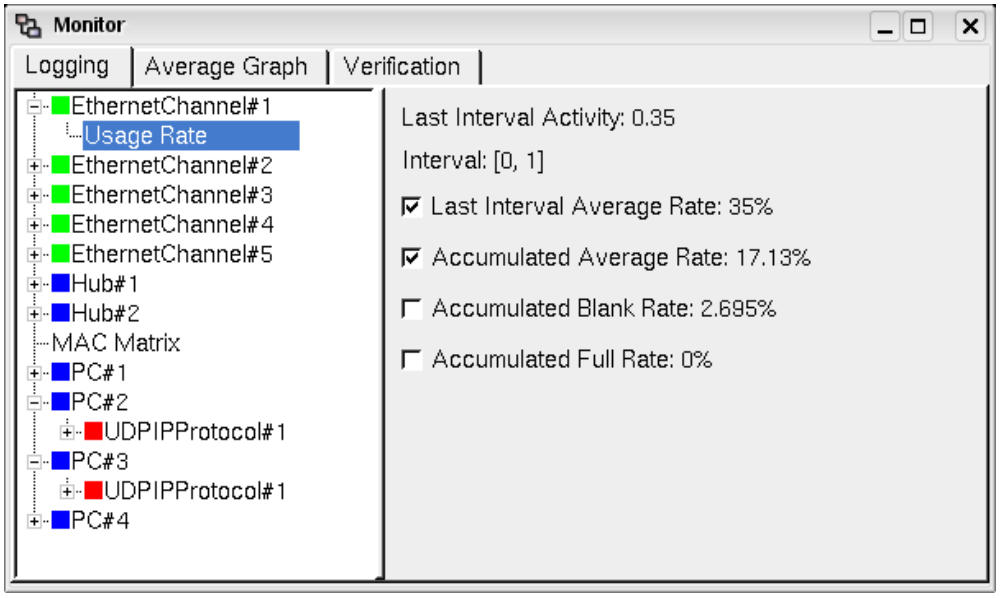

Abbildung 4.22: ESL-Target Monitor – Logging

werden. Die grafische Repräsentation zur Simulationszeit erfolgt durch den in Abbildung 4.22 dargestellten Performance-Monitor. Dieser beinhaltet alle Logging-Einträge und ordnet sie entsprechend der Block-Hierarchie bzw. architekturellen Topologie in eine Baumstruktur ein. Die verschiedenen Elementtypen des Baums wurden dabei zur leichteren Unterscheidung farblich ausgestaltet. Partitionen bzw. Ausführungseinheiten werden blau, Protokollstapel rot und Kanäle grün dargestellt. Zusätzlich erlaubt der Performance-Monitor die Selektion bestimmter Datenfelder zur Durchführung detaillierter Analysen und Vergleiche. Deren Visualisierung erfolgt durch den in Abbildung 4.23 gezeigten X-Y-Graph, wobei alle Datenfelder anhand eines Rasters normalisiert und gemeinsam dargestellt werden. Schlussendlich sei erwähnt, dass die Genauigkeit der Logging-Einträge vom Parameter `ESLSetup` und dem dort konfigurierten Update-Intervall abhängt.

Vergleicht man den Speicherverbrauch zur Simulationszeit bei zentraler Datenerfassung durch das ESL-Target, sowie dezentraler durch einzelne Blöcke, sind ähnliche Ergebnisse zu beobachten. Dies liegt darin begründet, dass prinzipiell die gleichen Daten zwischengespeichert werden. Der konkrete Speicherverbrauch hängt hierbei hauptsächlich von der gewählten Puffergröße ab, welche jedoch nur bei Verwendung des ESL-Targets zentral konfiguriert werden kann. Zudem vereinfacht das ESL-Target die synchronisierte Verarbeitung und Ausgabe der gewünschten Performance-Werte. In Bezug auf die

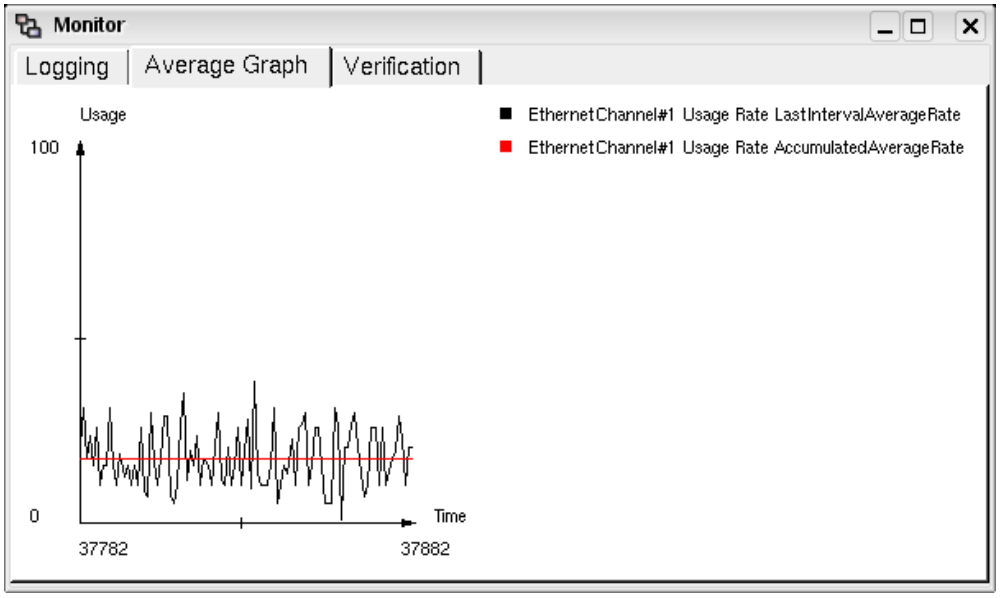

Abbildung 4.23: ESL-Target Monitor – AverageGraph

Simulationsgeschwindigkeit ist ebenfalls der Einsatz des ESL-Targets zur Datenerfassung vorteilhaft, da jedes Primitive einen Pointer auf dieses besitzt und somit eine direkte, performantere Datenhinterlegung möglich ist. Als Nachteil ist die Notwendigkeit zusätzlicher Primitives zur Kommunikation mit dem ESL-Target zu nennen, was eine leichte Vergrößerung des Modells zur Folge hat.

4.6.5 Architektur-Generator

Der Architektur-Generator *ArchGen* ermöglicht die automatisierte Erstellung architektureller Gesamtsystemmodelle S_{Arch}. Als Ausgangspunkt dienen funktionale Gesamtsystemmodelle S_{Func}, welche im Vorab durch architekturspezifische Konfigurationsinformationen $K_{ArchGen}$ zu erweitern sind. Demzufolge führt der Architektur-Generator eine Modelltransformation mit folgender Funktion durch:

$$f_{ArchGen}(S_{Func}, K_{ArchGen}) \rightarrow S_{Arch}$$

Anzumerken ist, dass ebenfalls nicht modellbasiert erfasste Spezifikationen als Ausgangspunkt möglich sind, jedoch im Rahmen der Arbeit unbetrachtet

bleiben. Um den Architektur-Generator verwenden zu können, muss dessen Quellcode in ein ausführbares Binary überführt werden. Folgende Schritte sind hierzu notwendig:

- Wechsel in das Verzeichnis `~/MLD/ArchitectureBlockSet/generator`, Aufruf des Befehls `make release` zur Übersetzung des Quellcodes
- Aufruf des Generators durch den Befehl `./generator`

Im weiteren Verlauf werden die Anforderungen an den Architektur-Generator erfasst und erläutert. Daraufhin wird auf die Konfiguration sowie Realisierung eingegangen.

Die Architektur von Systemen zeichnet sich durch eine Vielzahl an Eigenschaften aus, welche während der Generierung zu berücksichtigen sind. Da die gemeinsamen Eigenschaften bereits im Rahmen der Formalisierung aus Abschnitt 4.5 hergeleitet und in Modellelemente überführt wurden, können diese nun zur Erfassung der Anforderungen herangezogen werden. Im Ergebnis lassen sich drei Kategorien abstrahieren:

- *Topologische Anforderungen* beziehen sich auf Festlegung der Anzahl an Partitionen inklusive etwaig notwendiger Transitsysteme, sowie deren Verbindung durch Kanäle.
- Als *Komponenten-Anforderungen* bezeichnet man die Selektion und Parametrisierung von Architekturkomponenten, also Partitionen bzw. Ausführungseinheiten, Protokollstapeln und Kanälen.
- Unter *Partitionierungs-Anforderungen* versteht man die Verteilung der Funktionen auf Partitionen.

Zur Erfüllung der Anforderungen müssen MLDesigner-Modelle in einer bestimmten Art und Weise durch den Architektur-Generator erzeugt werden. Grundlage bildet die XML-basierte Datenhaltung der Modelle sowie die Umsetzung der formalisierten Architekturbeschreibung durch die Basisbibliothek und Architekturbibliothek.

Abbildung 4.24 zeigt den schematischen Ablauf zur Generierung eines Architekturmodells bzw. architekturellen Gesamtsystemmodells. Als Ausgangspunkt dient eine XML-basierte Konfigurationsdatei, welche mehrere reguläre Ausdrücke zur Steuerung der Generierung beinhaltet (siehe Anhang C). Dabei wird zwischen einer direkten und indirekten Informationserfassung unterschieden. Bei ersterer beinhaltet die Konfigurationsdatei die Informationen

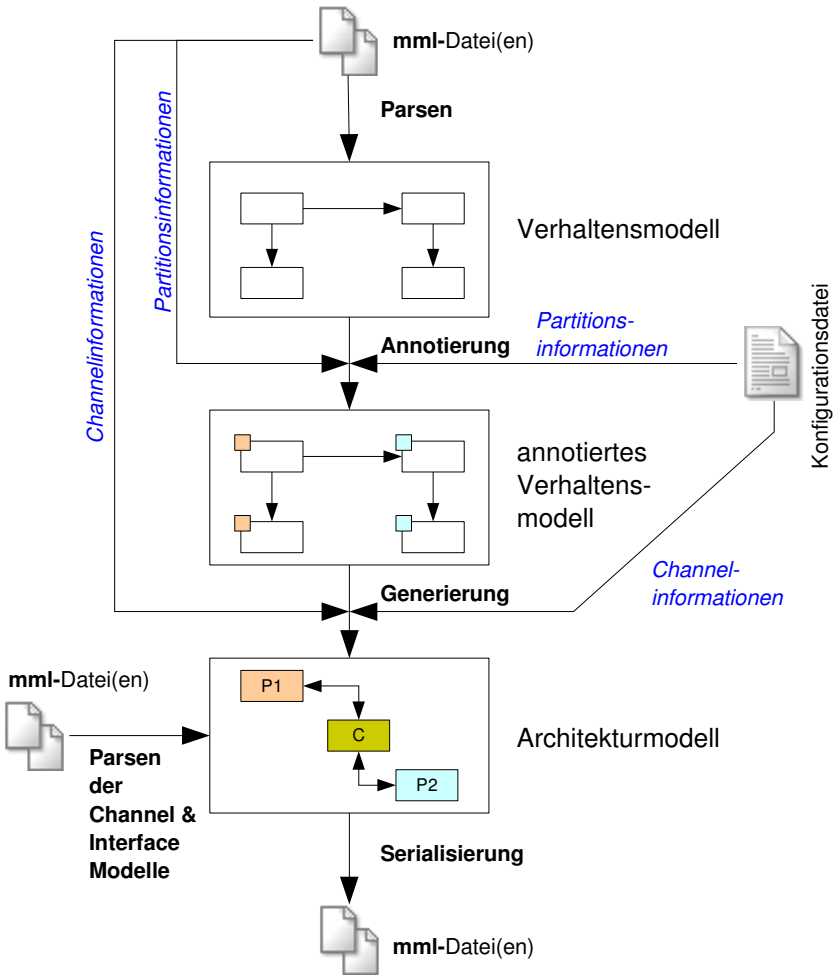

Abbildung 4.24: Ablaufplan zum Architektur-Generator [Rath 2007]

selbst, bei letzterer lediglich einen Verweis auf Annotierungen am funktionalen Gesamtsystemmodell. Hervorzuheben ist, dass die direkte Informationserfassung Voraussetzung zur Verwendung innerhalb eines ausführbaren Entwurfsprozesses ist, da nur so unterschiedliche Konfigurationsdateien erzeugt und ausgetauscht werden können. Kommt es zur Ausführung des Architektur-Generators wird die entsprechend zugeordnete Konfigurationsdatei geparst und ausgewertet. Diese enthält folgende Informationen:

- Pfad zum Verhaltensmodell bzw. funktionalen Gesamtsystemmodell auf dem Dateisystem

- Mappings funktionaler Submodelle auf Partitionen
- Ausführungseinheiten und Protokollstapel der Partitionen, sowie deren Pfade auf dem Dateisystem
- Netzwerke zwischen den Partitionen, umsetzende Kanäle und deren Pfade auf dem Dateisystem
- Pfad zur Abspeicherung des neu erstellten architekturellen Gesamtsystemmodells auf dem Dateisystem

Daraufhin wird das angegebene funktionale Gesamtsystemmodell geparst und in Form von Instanzen einer modellbeschreibenden Klassenstruktur im Speicher abgebildet. Erst jetzt kann die eigentliche Generierung vollzogen werden. Hierzu wird ein neues Modell im Speicher erzeugt und schrittweise unter Berücksichtigung der gewählten Konfiguration konkretisiert, sprich mit Modellelementen gefüllt:

- Anlegen von Modulen für die einzelnen Partitionen
- Einbindung der partitionierten funktionalen Submodelle in die Partitionen sowie der unpartitionierten in das Umgebungsmodell
- Integration von Protokollstapeln in die Partitionen
- Parametrisierung der Partitionen zur Zuordnung der Ausführungseinheiten und Adressierungsinformationen
- Verbindung der Partitionen durch Kanäle

Abschließend wird das im Speicher vorhandene Modell serialisiert. Im Ergebnis entstehen mehrere MLDesigner-Modelle, welche gemeinsam das architekturelle Gesamtsystemmodell beschreiben und am konfigurierten Pfad abgespeichert werden. Weiterführende Informationen zur Funktionsweise des Architektur-Generators sind in [Rath 2007] zu finden.

4.7 Erweiterung der Architekturbibliothek

Im folgenden wird die Vorgehensweise zur Erweiterung der in Abschnitt 4.6.3 eingeführten Architekturbibliothek um weitere Ausführungseinheiten, Protokollstapel und Kanäle erläutert. Jede dieser Komponenten besitzt eine standardisierte Schnittstelle, die beim Entwurf einzuhalten ist. Dabei kann die

Verhaltensbeschreibung an sich frei durch den Nutzer bestimmt werden, weswegen zunächst ein Abstraktionsniveau bestimmt werden muss. Die Wichtigkeit dieser Entscheidung ist nicht zu unterschätzen, da sie direkten Einfluss auf Modellgenauigkeit und Simulationsgeschwindigkeit hat, wodurch wiederum Simulationsergebnisse beeinflusst werden. Abbildung 4.25 stellt diesen Zusammenhang in Form eines Diagramms dar. Man erkennt, dass die Simulationsgeschwindigkeit mit der Modellgenauigkeit abnimmt.

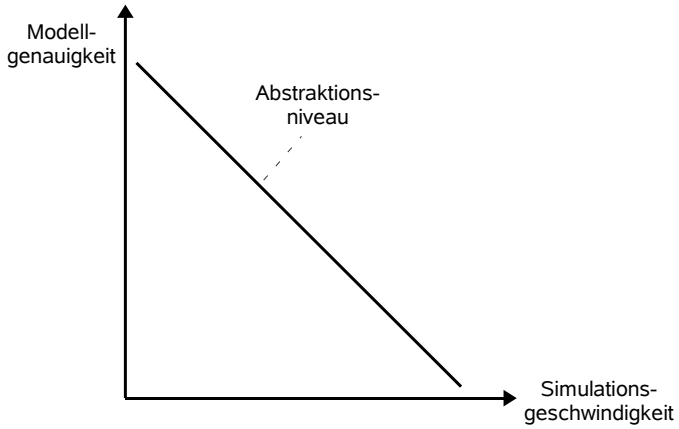

Abbildung 4.25: Bestimmung des Abstraktionsniveaus

Zur Festlegung eines geeigneten Abstraktionsniveaus, müssen die Anforderungen an das zu entwerfende Modell analysiert werden, um daraus die notwendigen Modelleigenschaften abzuleiten. In Bezug auf Ausführungseinheiten ist unter anderem zu entscheiden, ob eine Ressourcenverwaltung auf Betriebssystemebene mit Tasks und Prozessen oder auf Prozessorebene in Abhängigkeit von Geschwindigkeit, Instruction Set und Prozessor Pipeline eingesetzt werden sollte. Bei Protokollstapeln und Kanälen ist zu bestimmen, wie detailliert Bandbreiten, Datenfluss-Muster, Verfügbarkeiten und Übertragungsqualitäten zu erfassen sind. Soll beispielsweise lediglich die Bandbreite untersucht werden, reicht ein relativ einfaches Queue-Modell aus. Soll jedoch eine detaillierte Analyse der zu übertragenden Pakete durchgeführt werden, sind alle beteiligten Kommunikationsprotokolle zu beschreiben. Im letzteren Fall müssen daher mehrere Queues bzw. Puffer eingesetzt werden. Abbildung 4.26 stellt beide Varianten hinsichtlich der Datenfragmentierung gegenüber. Es ist leicht zu erkennen, dass durch das im oberen Teil gezeigte abstraktere Modell nicht die gesamte Kommunikationsdynamik erfasst werden kann. Eine ver-

gleichbare Problematik ist auch beim Entwurf anderer Modellkomponenten zu beobachten. Beispielsweise können Funktionen detailliert oder als abstrakte, probabilistische Traffic-Generatoren beschrieben werden [Below 2003].

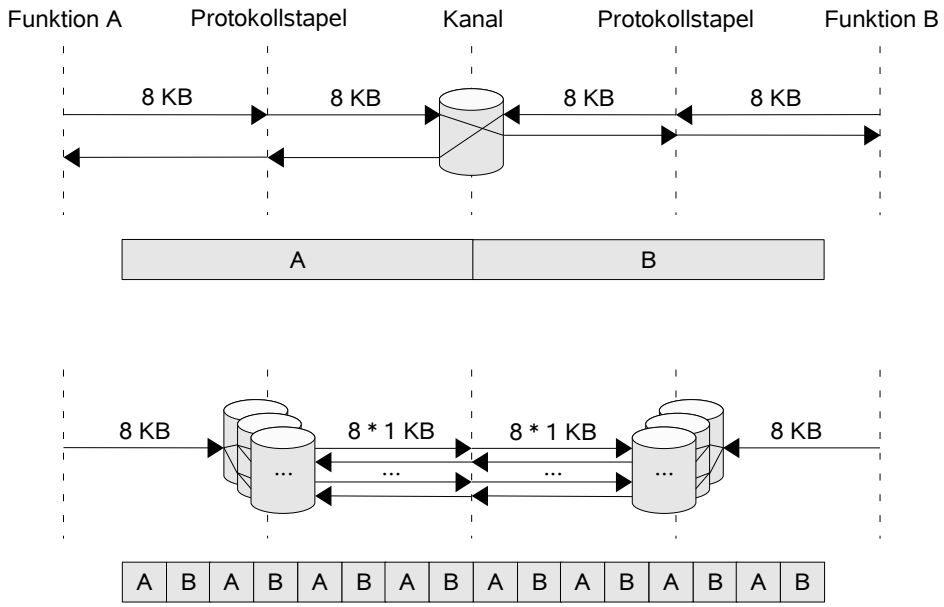

Abbildung 4.26: Abstraktionsniveauabhängige Datenfragmentierung

Der eigentliche Entwurf, genauer gesagt die Modellerstellung, kann entweder auf Basis verfügbarer Spezifikationen oder anhand real existierender Systeme erfolgen. Ziel ist es, die relevanten Performance-Eigenschaften in ein ausführbares Modell zu überführen und dieses im Anschluss zu validieren. Eine Ermittlung am realen System ist anhand mehrerer Methoden möglich, wobei zwischen Realisierungen in Hard- und Software zu unterscheiden ist. Um Software-Performance zu bemessen, können statistische Algorithmen, quellcodebasierende Analysen, compilerbasierende Analysen, sowie Instruction Set Simulatoren eingesetzt werden [Lohfelder 2004]. Zur Bemessung von Hardware-Performance können unter anderem Funktionsaufrufzeiten anhand Oszilloskop-basierter Pin-Analysen ermittelt werden. Eine weitere Möglichkeit bietet die Programmierung von Timern auf der zu analysierenden Hardware mit anschließender Laufzeitausgabe über eine Schnittstelle. Zudem können auch VHDL-Simulatoren eingesetzt werden [Zens 2003].

Die entworfenen Architekturkomponenten bestehen im Normalfall aus mehreren Submodellen, welche hauptsächlich die notwendigen Protokolle repräsentieren. Außerdem existieren oftmals Modelle zur Validierung, sowie Beispiele zur Anwendung der Architekturkomponenten. All diese Modellkomponenten sind in bestimmte MLDesigner-Bibliotheken eingegliedert. Hierbei ist es von Vorteil eine einheitliche Strukturierung einzuhalten, um einen Wiedererkennungseffekt zur beschleunigten Anwendung zu erreichen. Da eine solche Strukturierung bereits durch die aktuellen Protokollfamilien der Architekturbibliothek verwendet wird, sollte diese auch durch alle neu hinzugefügten eingesetzt werden. Aus diesem Grund wird eine Einteilung in folgende vier Subbibliotheken vorgeschlagen:

- Die Bibliothek *Architecture* beinhaltet unterschiedliche Ausprägungen der Architekturkomponenten.
- Innerhalb der Bibliothek *Components* werden die notwendigen Subsysteme abgelegt.
- Alle zur Validierung der Architekturkomponenten erstellten Modelle sind Bestandteil der Bibliothek *Validation*.
- Beispiele hinsichtlich Integration und Parametrisierung der Architekturkomponenten befinden sich in der Bibliothek *Examples*.

4.8 Performance-Analyse

Im Rahmen des vorliegenden Abschnitts wird der Einsatz der Anwendungsumgebung zur Durchführung von Performance-Analysen in den frühen Entwurfsphasen erläutert. Zunächst werden einige Grundlagen eingeführt.

Definition 16: Unter einer *Performance-Analyse* versteht man die Untersuchung des Verhaltens eines Systems bzw. Systemmodells anhand von Informationen, die während dessen Ausführung gesammelt werden.

Ziel ist es, den erstellten Entwurf gegen zuvor definierte funktionale und nichtfunktionale Anforderungen zu validieren. Unter anderem können folgende Fragen beantwortet werden:

- Ist die Systemfunktionalität unter Auslastung adäquat erfüllbar?
- Verhalten sich alle Systemkomponenten unter Auslastung wie erwartet?

- Welche Systemkomponenten bremsen das System aus (Bottlenecks) und was sind die Ursachen hierfür?
- Müssen bzw. können einige Systemkomponenten verbessert werden?
- Sind die Ressourcen des Systems ausreichend dimensioniert?
- Ist die gewählte Netzwerktopologie geeignet?

Hierzu ist es notwendig, die Systemfunktion, -architektur und -umgebung in Form eines Gesamtsystemmodells zu erfassen, um dieses simulativ zu analysieren [Baumann u. a. 2007b]. Die ermittelten Informationen können anschließend zur Dimensionierung des Systems verwendet werden. Folgende Übersicht benennt und erläutert solche Informationen:

- Als *Durchsatz* bezeichnet man die Menge oder Nutzlast an Datenpaketen, die pro Zeiteinheit verarbeitet bzw. übertragen wird. Bildet man den Quotienten zwischen aktuellem und maximalem Durchsatz, erhält man die sogenannte Netzlast in Prozent. Des weiteren spricht man von einer Sättigung, wenn zu viele Datenpakete versendet werden, so dass der maximale Durchsatz überschritten wird und nicht mehr alle Datenpakete beim Empfänger ankommen. Sowohl Durchsatz als auch Netzlast werden oftmals in Form einer Matrix der beteiligten Netzwerkteilnehmer veranschaulicht.
- Unter *Verzögerung* versteht man die Zeit, die ein Datenpaket zur Übertragung von einem Sender zu einem Empfänger benötigt. Umso höher der Datenverkehr ist, desto höher wird die Verzögerung und damit auch die Reaktionszeit. Ursache hierfür ist die Auslastung des verbindenden Kanals und die sich daraus ergebende Notwendigkeit zur Zwischenspeicherung der Daten. Oftmals werden auch Durchschnittsverzögerungen ermittelt, das heißt ein Mittelwert über einen bestimmten Zeitraum gebildet. Zudem können Sende- und Empfangs-Jitter überwacht werden, um die Varianz der Verzögerungen bzw. Durchlaufzeiten von Datenpaketen durch das Netzwerk zu erfassen.
- Die *Ressourcenauslastung* beschreibt den Einsatz von Betriebsmitteln zur Ausführung von Systemfunktionen. Dabei wird zwischen quantitativen Ressourcen und Server-Ressourcen unterschieden. Erstere besitzen eine Mengendimension, die teilweise oder ganz durch Systemfunktionen alloziert werden kann, letztere eine Queue sowie Scheduling-Strategie zur Verwaltung funktionaler Anfragen. Beispiele für Ressourcen sind

Sende-Puffer von Protokollen, Übertragungsmedien, CPU-Speicher und CPU-Prioritäten.

- Unter *Datenfragmentierung* versteht man die Aufspaltung zu verarbeitender bzw. zu übertragender Datenpakete in mehrere kleine Datenpakete. Umso höher die Anzahl ist, desto höher ist auch der Protokoll-Overhead. Die Datenfragmentierung kann durch Konfiguration der beteiligten Protokolle beeinflusst werden.

- Als *statistischer Fehler* wird das probabilistisch gesteuerte Eintreten einer Abweichung vom Normalzustand bezeichnet. Beispiele hierfür sind ein Verlust zu übertragender Datenpakete oder fehlerhafte Speicherzugriffe. In diesem Zusammenhang wird unter Fehlertoleranz die Fähigkeit des Systems verstanden, auch bei eingetretenen Fehlern funktionstüchtig zu bleiben. Als weiterer wichtiger Messwert ist die Bit Error Rate (BER) zu nennen, durch welche die Fehleranzahl pro Zeiteinheit beschrieben wird.

Zur Informationsermittlung können *Queuing-Methoden* sowie *probabilistische Methoden* eingesetzt werden [Dally u. Towles 2004]. Bei Queuing-Methoden werden durch Quellen Datenpakete mit einer bestimmten Rate generiert und in einer Queue abgelegt. Diese werden wiederum mit einer bestimmten Rate von Servern abgeholt. Quellen und Server werden durch Funktionen und Kanäle repräsentiert. Der Zustand der Queue, also die Anzahl der enthaltenen Elemente, kann durch Markov-Ketten dargestellt werden. Im Gegensatz dazu verwenden probabilistische Methoden keine Queues. Hier werden Verzögerungen anhand von durchschnittlichen Wartezeiten berechnet. Die im Rahmen der Arbeit entwickelte Anwendungsumgebung unterstützt beide Varianten (siehe auch Abschnitt 4.7).

Die Durchführung von Performance-Analysen in MLDesigner beruht auf dessen Simulationsmöglichkeiten. Dabei wird der Datenfluss zwischen sowie innerhalb der standardisierten Architekturkomponenten zur Ausführungszeit überwacht und ausgewertet. Hierdurch können Paketverzögerungen, -mengen und -größen, sowie Netzlasten erfasst werden. Zur Veranschaulichung kommt der Performance-Monitor zum Einsatz, wobei mehrere Visualisierungsarten zur Verfügung stehen. Die wichtigsten werden in den Abbildungen 4.22, 4.23 und 4.27 dargestellt.

Wird die Simulation gestartet, beginnt damit auch die Performance-Analyse. Dies bedeutet, dass jede Architekturkomponente die empfangenen Daten entsprechend ihrer Eigenschaften beeinflusst, sprich verzögert, fragmentiert, mo-

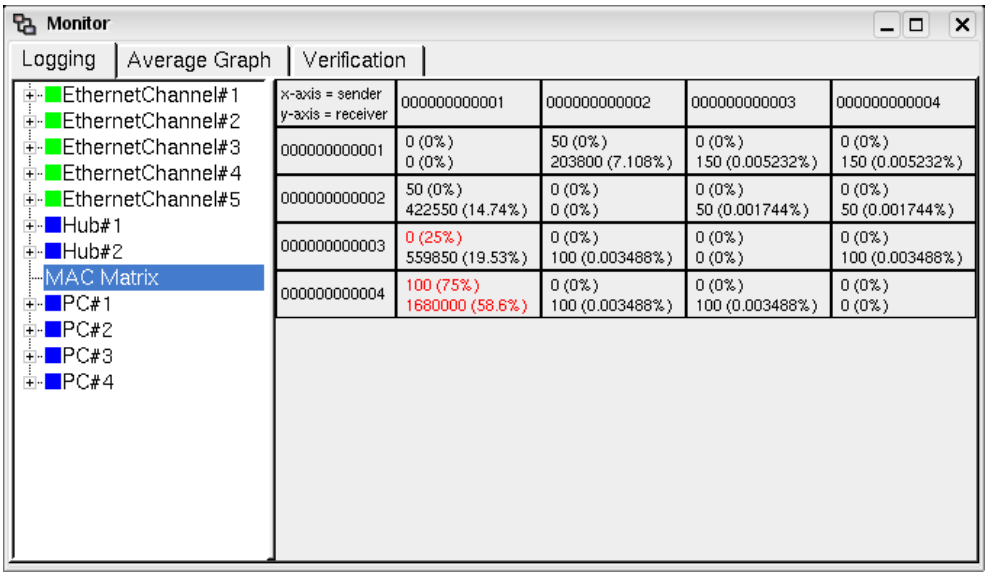

Abbildung 4.27: ESL-Target Monitor – Netzlast-Matrix

difiziert, verwirft oder dupliziert. Gleichzeitig werden die definierten Ressourcen alloziert bzw. freigegeben. Abbildung 4.28 veranschaulicht diesen Vorgang am Beispiel zweier Partitionen (p_1, p_2), die über einen Kanal (c_1) verbunden sind. Das zugehörige Blockflussdiagramm wird im oberen Teil von Abbildung 4.17 dargestellt. Jede Partition besitzt eine Ausführungseinheit zur Ausführung der zugeordneten Funktionen und Protokollstapel (f_1, f_2, ps_1, ps_2). Diese wiederum besitzen eine Priorität und einen Speicherbedarf. Folglich kann es während der Simulation zu Task-Wechseln und damit zu zusätzlichen Verzögerungen sowie einer erhöhten Speicherauslastung kommen. Des weiteren ist jeder Kanalzugriff mit einer Fragmentierung in mehrere Pakete verbunden, wodurch bei gleichzeitigen Zugriffen zusätzliche Verzögerungen entstehen. Dabei ist anzumerken, dass es infolge von Datenabhängigkeiten und architekturellen Verzögerungen zu Blockierungen beim Nachrichtenaustausch kommen kann. Möglichkeiten zur Überwachung solcher kritischen Nachrichten werden in [Pacholik u. Fengler 2007] erläutert.

Um eine erfolgreiche Performance-Analyse durchführen zu können, müssen bestimmte Voraussetzungen erfüllt werden. Zunächst ist ein geeignetes Abstraktionsniveau festzulegen. Hierfür ist es notwendig entsprechend detaillierte Architekturkomponenten zu entwerfen. Diese müssen validiert und bei Verwendung korrekt parametrisiert werden. Des weiteren ist ein Umgebungsmodell zu entwerfen, welches die zu untersuchenden Missionen bzw. Testfälle

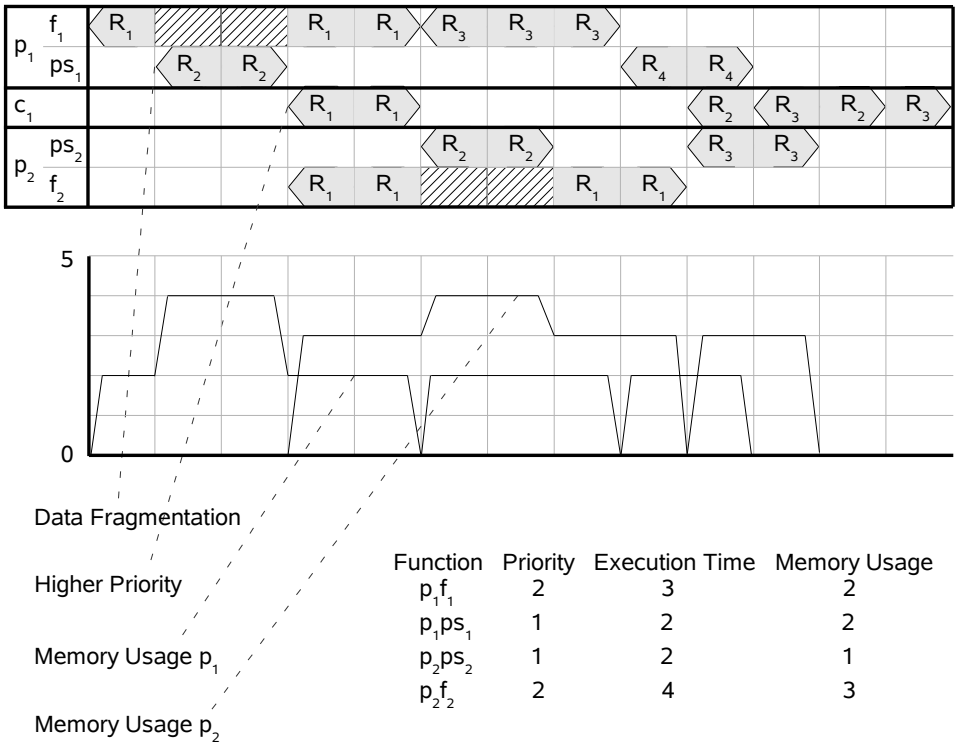

Abbildung 4.28: Performance-Analyse am Beispiel

enthält. Letztlich sei noch erwähnt, dass eine ausreichend lange Simulationszeit festzulegen ist, da nur so alle verzögerten Datenpakete das Netzwerk komplett durchlaufen können.

4.9 Grenzen der Anwendungsumgebung

Während der Realisierung der Anwendungsumgebung wurden mehrere Probleme aufgefunden. Diese ergeben sich aus den spezifischen Eigenschaften von MLDesigner und beziehen sich auf die Erstellung von Architekturkomponenten, sowie die Ausführung und Generierung realistischer Gesamtsystemmodelle. Im weiteren Verlauf werden die Probleme aufgeführt und zum Teil Lösungsvorschläge unterbreitet.

Beim Entwurf und der Validierung von Architekturkomponenten muss ein Abstraktionsniveau festgelegt werden. Daraus folgt, dass jede Architekturkomponente lediglich in einem bestimmten Kontext sinnvoll anwendbar ist. Bei Nichtbeachtung kann es zur Ermittlung inkorrekter Simulationsergebnisse und somit auch Performance-Werte kommen. Dabei ist zu beachten, dass die architekturellen Eigenschaften im großen Maße von ihrer Implementation in Hard- und Software abhängen und oftmals schwer bestimmbar sind. Ein weiteres Problem besteht darin, dass das zur Ausführung benötigte ESL-Target stets durch die oberste Hierarchieebene des Modells, sowie alle eingebetteten DE-Modelle verwendet werden muss. Letztere sind daher mit dem Target `<parent>` zu konfigurieren, was jedoch gleichzeitig die Verwendung anderer nutzerspezifischer Targets unmöglich macht. Durch eine direkte Anbindung der ESL-Target-Funktionalität an MLDesigner könnte dies vermieden werden.

Auch während der Modellausführung können mehrere Probleme auftreten. Da die standardisierten Architekturkomponenten unter anderem auf kompatiblen Datenstrukturen beruhen und diese durch MLDesigner lediglich syntaktisch jedoch nicht semantisch überwacht werden können, existiert an dieser Stelle eine potenzielle Fehlerquelle. Außerdem können zur Simulationszeit Seiteneffekte zwischen Blöcken auftreten und zu einer Verfälschung der Analysedaten führen. Ursache hierfür ist die indirekte Kommunikation über Pointer und Argumente innerhalb von Primitives, wodurch die direkte, portbasierte Kommunikation der standardisierten Architekturkomponenten übergangen wird. Unter Argumenten versteht man in diesem Zusammenhang die MLDesigner-Komponenten Memory, Resource und Event. Eine Lösung des Problems ist aktuell nicht möglich, da die indirekte Kommunikation nicht überwacht werden kann. Zudem widerspricht diese grundsätzlich dem Modellierungskonzept gekoppelter Systeme [Grüner 2007]. Weitere Probleme resultieren aus der dynamischen Integration von Ausführungseinheiten. Einerseits kann eine Abänderung der Scheduling-Reihenfolge auftreten, was vor allem aus der nicht vorhandenen Unterscheidung zwischen Und- sowie Oder-Verknüpfungen beim Datenempfang folgt. Andererseits werden die zu verzögernden Blöcke erst ausgeführt und dann verzögert, wodurch die Seiteneffekt-Problematik noch verstärkt wird.

Bei Anwendung des Architektur-Generators kann es ebenfalls zu modellbedingten Problemen kommen. Unter anderem stellt sich die Generierung auf Grundlage von Modellen mit Auto-Forks und Auto-Merges als problematisch dar. Durch diese wird der Datenfluss geteilt bzw. zusammengeführt, wie im oberen Teil von Abbildung 4.29 zu erkennen ist. Da beide keine Block-

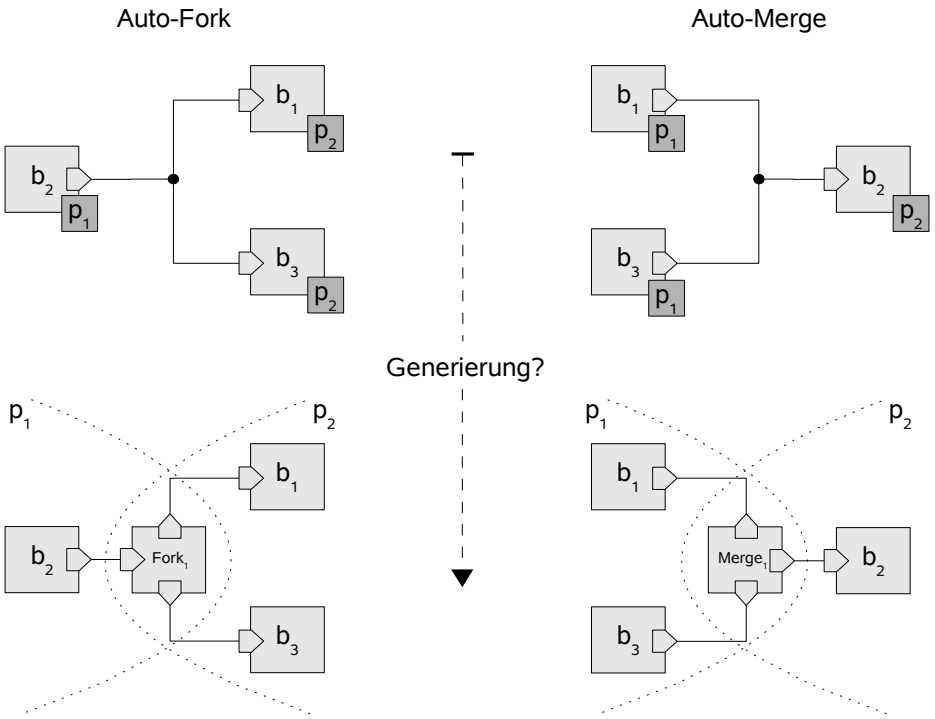

Abbildung 4.29: Partitionierungsproblem bei Auto-Forks und Auto-Merges

Repräsentation besitzen, ist auch keine Zuordnung auf zu generierende Partitionen möglich. Daraus folgt, dass nicht eindeutig bestimmt werden kann, auf welcher Partition der Datenfluss geteilt bzw. zusammengeführt wird, was jedoch großen Einfluss auf die Menge der zu übertragenden Daten und somit die System-Performance hat. Zur Lösung des Problems müssen vor der Generierung alle Auto-Forks durch Fork-Blöcke und alle Auto-Merges durch Merge-Blöcke ersetzt werden. Daraufhin ist eine Zuordnung der eingefügten Blöcke auf Partitionen möglich. Das entstehende System wird im unteren Teil der Abbildung dargestellt. Ein weiteres Problem des Architektur-Generators ergibt sich aus der Verwendung verlinkter Argumente. Im oberen Teil von Abbildung 4.30 wird deren Funktionsweise dargestellt. Die gestrichelten Linien veranschaulichen die portlose Kommunikation zweier Blöcke über ein Argument. Dabei sendet ein Block Daten an das Argument, während ein weiterer diese von dort empfängt. Da zur Durchführung von Performance-Analysen der Datenfluss über die portbasierten, standardisierten Architekturkompo-

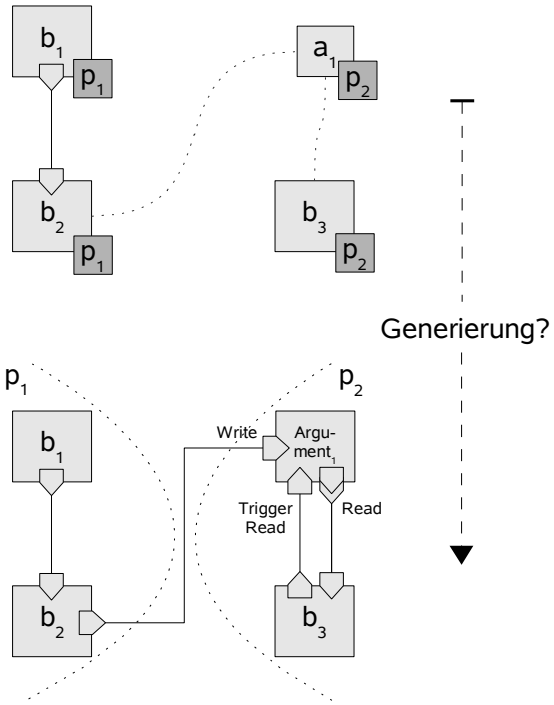

Abbildung 4.30: Partitionierungsproblem bei Argumenten

nenten verlaufen muss, ist es notwendig die verlinkten Argumente vor der Generierung zu ersetzen. Ein Lösungsvorschlag wird im unteren Teil der Abbildung dargestellt. Man erkennt, dass das Argument durch einen Block ersetzt wurde, welcher drei Ports besitzt. Über den Port `Write` kann ein neuer Wert auf das Argument geschrieben werden. Anhand des Ports `TriggerRead` ist es möglich den aktuellen Wert anzufordern, welcher daraufhin über den Multiport `Read` an den jeweils anfragenden Block versendet wird. Demzufolge kann die gesamte indirekte Kommunikation in eine direkte Kommunikation überführt werden. Anzumerken ist, dass zusätzlich noch ein vierter Port eingeführt werden kann, um alle lesenden Blöcke zu benachrichtigen wenn neue Daten anliegen.

Die genannten modellbedingten Probleme können prinzipiell gelöst werden. Hierzu ist eine komplette Integration der Anwendungsumgebung in MLDesigner, sowie eine Überarbeitung des Blockflussdiagramms notwendig.

Kapitel 5

Anwendungsbeispiele

5.1 TCP/IP-Ethernet

5.1.1 Einführung

Der vorliegende Abschnitt beschäftigt sich mit der Anwendung des Architecture Block Sets für den Entwurf eines Protokollstapels sowie eines Kanals am Beispiel der Protokollfamilie von TCP/IP-Ethernet [Eck 2007]. Diese besteht grundsätzlich aus dem verbindungsorientierten Transmission Control Protocol (TCP), dem verbindungslosen Internet Protocol (IP) und dem als Local Area Network (LAN) klassifizierbaren Ethernet. Bezogen auf das ISO/OSI

ISO/OSI	**TCP/IP**	**Protokoll**
Bitübertragungsschicht	Netzzugangsschicht	Ethernet, CSMA-CD
Sicherungsschicht	~	~
Netzwerkschicht	Vermittlungsschicht	IP, ARP
Transportschicht	Transportschicht	TCP
Sitzungsschicht	Anwendungsschicht	—
Darstellungsschicht	~	—
Anwendungsschicht	~	—

Tabelle 5.1: Referenzmodelle und Protokolle bei TCP/IP-Ethernet

Referenzmodell aus Abbildung 4.5 unterstützt das TCP/IP Referenzmodell lediglich vier Schichten. Tabelle 5.1 stellt beide Referenzmodelle gegenüber und ordnet die im Rahmen des Beispiels verwendeten Protokolle ein. Im wei-

teren Verlauf werden die Protokolle zunächst schichtenorientiert erläutert, im Anschluss umgesetzt und letztlich validiert.

5.1.2 Protokollspezifikationen

Die Beschreibung der Transportschicht basiert auf der verbindungsorientierten und somit zuverlässige Protokollspezifikation TCP. Dabei handelt es sich prinzipiell um eine Voll-Duplex-Netzwerkverbindung zwischen zwei Endpunkten, über welche Informationen in beide Richtungen zugleich übertragen werden können. Jeder Endpunkt wird durch einen Socket, bestehend aus einer IP-Adresse und einem Port, definiert. Einer der wichtigsten Mechanismen von TCP ist die Überlastkontrolle. Diese ermöglicht die Steuerung der Übertragungsrate durch Begrenzung der Anzahl von gesendeten, jedoch noch nicht bestätigten TCP-Paketen bzw. Segmenten. Die Anzahl der zulässigen unbestätigten Segmente wird hierbei als TCP-Fenstergröße bezeichnet. Ziel ist es, schnellstmöglich eine größtmögliche Anzahl anstehender unbestätigter Segmente zu übertragen, solange keine aufgrund von Überlast verloren gehen. Im Verlustfall reduziert die TCP-Verbindung die Übertragungsrate auf ein sicheres Maß und beginnt anschließend erneut mit dem Abtasten auf eine mögliche ungenutzte Bandbreite.

Die Netzwerkschicht wird durch die Protokollspezifikationen IP und Address Resolution Protocol (ARP) beschrieben. Hauptaufgabe von IP ist einerseits die Adressierung inklusive Routing und andererseits die Fragmentierung der zu übertragenden Daten. Da IP zur Gruppe der verbindungslosen Protokollspezifikationen gehört, kann eine fehlerfreie Durchführung von Übertragungen nicht überprüft und demzufolge auch nicht gewährleistet werden. Anzumerken ist, dass im vorliegenden Beispiel lediglich eine bidirektionale Verbindung aufgebaut wird und Übertragungen an mehrere Empfänger bzw. Fehlermeldungen von Netzwerkknoten nicht auftreten können, weswegen die Protokollspezifikationen Internet Control Message Protocol (ICMP) und Internet Group Management Protocol (IGMP) unberücksichtigt bleiben. Außerdem bezieht sich das Beispiel auf das Internet Protocol Version 4 (IPv4), welches der de facto Standard zur Kommunikation über das Internet ist. Grundlage der Datenübertragung bilden sogenannte Ethernet-Frames. Diese beinhalten unter anderem die zu sendenden Daten und die Hardwareadresse bzw. Media Access Control (MAC) Adresse des Empfängers. Da sich MAC-Adressen (48 Bit) in Bezug zu logischen IPv4-Netzwerkadressen (32 Bit) durch eine höhere Bitanzahl auszeichnen, ist eine Erfassung durch IP-Pakete unmöglich. Daraus folgt, dass die MAC-Adresse des empfangenden Netzwerkteilnehmers vor dem

eigentlichen Sendevorgang zu ermitteln ist. In diesem Zusammenhang kommt die Protokollspezifikation ARP zum Einsatz. Dabei wird ein Broadcast-Paket mit der IP-Adresse des gesuchten Netzwerkteilnehmers versendet. Empfängt der entsprechende Netzwerkteilnehmer das Broadcast-Paket, antwortet er mit einem neuen Paket, welches die gesuchte MAC-Adresse beinhaltet. Parallel dazu aktualisieren sämtliche das Broadcast-Paket empfangenden Netzwerkteilnehmer ihren ARP-Cache, sprich die Zuordnungen von Absender-IP zu Absender-MAC.

Zur Beschreibung der Bitübertragungsschicht sowie der Sicherungsschicht werden die Protokollspezifikationen Ethernet und Carrier Sense Multiple Access - Collision Detection (CSMA-CD) verwendet. Hierbei ist Ethernet als eine kabelgebundene Übertragungstechnik zu verstehen und definiert Vorschriften bezüglich der Eigenschaften physikalischer Übertragungsmedien. Diese beziehen sich hauptsächlich auf Kabel- und Steckertypen, Datencodierungs- bzw. Signalisierungsarten, sowie Paketformate und Protokolle. Des weiteren wird zwischen den Betriebsarten Halb- und Voll-Duplex unterschieden. Bei Halb-Duplex greifen alle Teilnehmer auf ein gemeinsames Medium zu, mit der Einschränkung, dass stets nur ein Teilnehmer senden darf. Die dabei durchzuführende Zugriffssteuerung beruht auf dem Protokoll CSMA-CD, durch welches das Medium vor jedem Sendezugriff auf seine Verfügbarkeit überprüft und eine Kollisionserkennung während des Sendevorgangs durchgeführt wird. Bei Voll-Duplex kommen separate Übertragungsmedien, also mehrere Adern, zum Einsatz und ein gleichzeitiges Senden bzw. Empfangen durch beide Teilnehmer ist möglich. Beispiele für Ethernet-Netzwerke sind unter anderem über Koaxial-, Twisted-Pair- oder Glasfaserkabel betriebene Bus- und Stern-Topologien.

5.1.3 Implementation

Die Umsetzung der oben genannten Protokollspezifikationen erfolgt in Form von modellbasierten, ausführbaren Architekturkomponenten. Als Grundlage dienen die standardisierten Entwurfsmuster aus Abschnitt 4.6.3. Dabei sind sämtliche am Übertragungsprozess beteiligten Protokolle inklusive der spezifischen Paketverzögerungen, -fragmentierungen sowie -verluste zu berücksichtigen. Nur so können Performance-Analysen in Bezug auf paketvermittelte Netzwerke brauchbare Ergebnisse liefern.

Anhand des Protokollstapels `TCPIPProtocol` werden die Protokollspezifikationen aller Schichten, mit Ausnahme der Bitübertragungsschicht, implementiert und zusammengefasst. Abbildung 5.1 stellt die Bestandteile sowie deren

Beziehungen untereinander dar und benennt gleichzeitig die entsprechenden ISO/OSI Schichten. Ausgangspunkt bildet der standardisierte Block `ProtocolFunctionInterface`, welcher die zu übertragenden Anwendungsdaten in Datenstrukturen verpackt und an die nächste Schicht bzw. den nächsten Block weiterleitet.

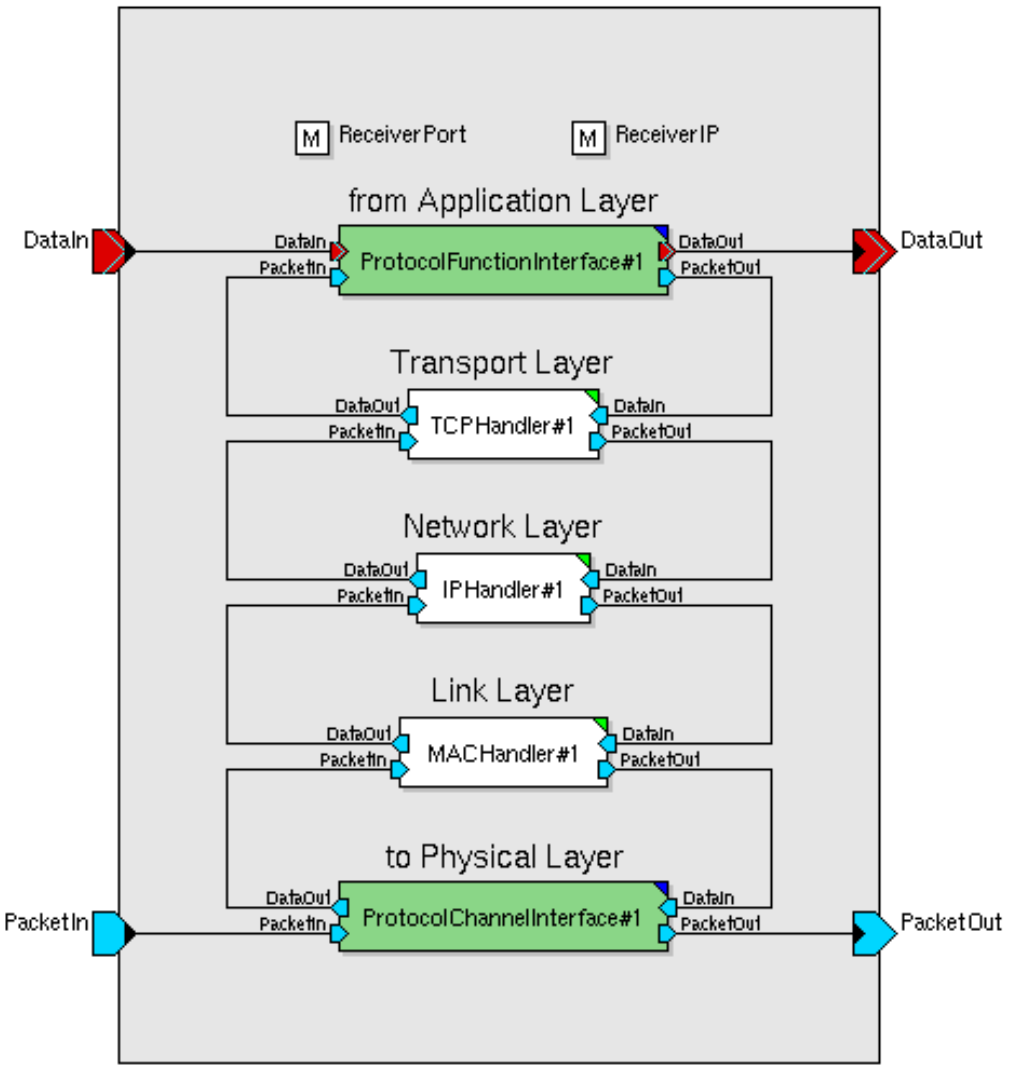

Abbildung 5.1: Protokollstapel TCPIPProtocol

Dabei handelt es sich um den Block `TCPHandler`, welcher aus den Blöcken `RenoSender` und `Receiver`, sowie dem Memory `ReceiverPort` besteht. Letzterer speichert während der Simulation den Zielport des Empfängers. Der `Re-`

noSender besteht wiederum aus weiteren Blöcken und ist für die Fragmentierung der zu übertragenden Daten, die Überlastkontrolle bzw. den Sliding Window Mechanismus, die Flusssteuerung, sowie den Verbindungsaufbau und -abbau zuständig. Auf Empfängerseite ist der Receiver dafür verantwortlich, die Bestätigungen der Pakete zu generieren, den Verlust von Paketen festzustellen und die empfangenen Daten an die Anwendungsschicht weiterzuleiten. Der TCPHandler kann anhand mehrerer Parameter konfiguriert werden. MMS determiniert die maximale Segmentgröße, WindowsSender die maximale Segmentgröße des Senders, WindowsReceiver die maximale Segmentgröße des Empfängers, initTime den initialen Wert des Retransmission Timer, Rate die minimale Pausenlänge zwischen Übertragungen, ackTimer die maximale Zeitdifferenz zwischen zwei Bestätigungspaketen und applicationDelay die maximale Leserate bezüglich der Paketabholung durch die Anwendungsschicht.

Im nächsten Schritt gelangen die Pakete zum Block IPHandler, welcher aus den Blöcken IPDatagramHandler und ARPLogic besteht. Ersterer kontrolliert den Übergang von bzw. zur Transportschicht und vollzieht gleichzeitig die IP-Adressierung der Pakete. Hierdurch wird eine Kommunikation zwischen Absender-IP und Empfänger-IP über das gesamte Kommunikationsnetz ermöglicht. Letzterer verwaltet die Abbildung von Netzwerkadressen auf Hardwareadressen, über welche später die Adressierung innerhalb der Sicherungsschicht erfolgt. Die eigene IP-Adresse kann anhand des Parameters IP festgelegt werden.

Anschließend erreichen die Pakete den Block MACHandler, dessen wichtigste Komponenten die Blöcke FrameBuilder, FIFOQueue, CSMA_CD und FrameReceiver sind. Aufgabe des MACHandler ist es, Ethernet-Pakete zu erstellen sowie im Halb-Duplex-Betrieb den Zugriff auf das Medium zu steuern. Zur Erstellung von Ethernet-Paketen müssen diese mit der Hardware-Adresse des Senders und des Empfängers versehen werden. Um letztere zu erhalten, versendet der Block FrameBuilder eine Anfrage an die Netzwerkschicht, welche daraufhin zeitlos mit der entsprechenden Information antwortet. Wie bereits oben erwähnt, unterscheidet Ethernet zwischen den Betriebsarten Voll- und Halb-Duplex. Bei Voll-Duplex werden die Ethernet-Pakete direkt zum ProtocolChannelInterface weitergeleitet, bei Halb-Duplex zuvor in die Warteschlange FIFOQueue eingereiht und durch den Block CSMA_CD abgearbeitet. Für den Fall, dass Pakete an den Localhost oder die eigene IP-Adresse adressiert sind, übergibt der FrameBuilder die Pakete direkt an den FrameReceiver. Werden umgekehrt Pakete vom Kanal bzw. ProtocolChannelInterface empfangen, kommt ebenfalls der FrameReceiver zum Einsatz. Dabei

wird zum einen überprüft, ob die Pakete an die eigene MAC-Adresse gerichtet sind und zum anderen, ob es sich um ein Broadcast-Paket handelt. Bei erfolgreicher Überprüfung werden die Daten extrahiert und an die Netzwerkschicht gereicht. Zusätzlich wird jedes Paket dem Block `CSMA_CD` gemeldet und auf ein Jamming-Signal untersucht. Die Konfiguration des `MACHandler` erfolgt durch die Parameter `MAC`, `EthernetVersion`, `EthernetImplementation` und `Operation`, wobei der erstgenannte die eigene MAC-Adresse und der letztgenannte die Betriebsart determiniert.

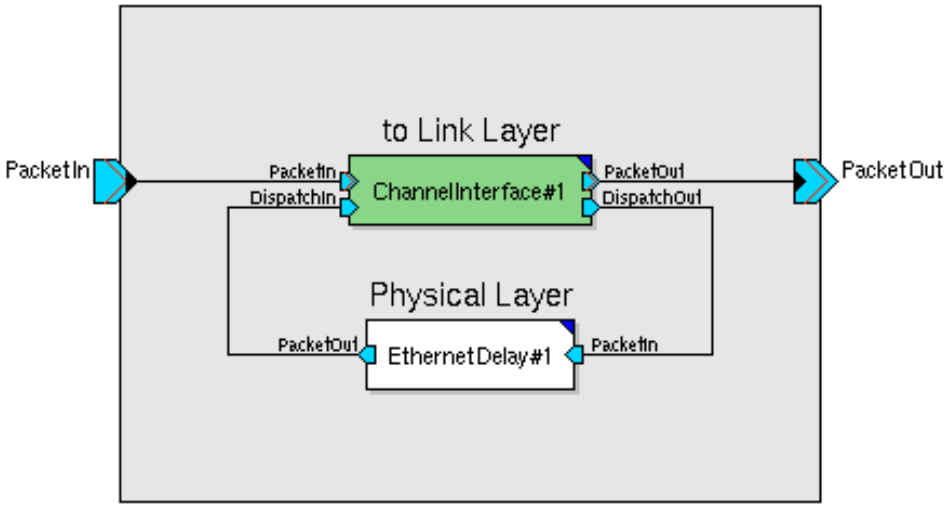

Abbildung 5.2: Kanal EthernetChannel

Die Eigenschaften von Ethernet werden durch das in Abbildung 5.2 dargestellte Kanalmodell `EthernetChannel` umgesetzt. Dieses besteht aus dem standardisierten Block `ChannelInterface` und dem kanalspezifischen Block `EthernetDelay`. Letzteres beschreibt die Übertragungsraten sowie Signalausbreitungsgeschwindigkeiten der unterschiedlichen Kabeltypen und bestimmt parallel die jeweiligen Paketverzögerungen. Mittels der Parameter `CableType` und `CableLength` kann das Kanalmodell konfiguriert werden. Folgene Formel dient als Berechnungsgrundlage:

$$Verzögerung = \frac{Kabellänge}{Ausbreitungsfaktor * Lichtgeschwindigkeit} + \frac{Paketgröße}{Übertragungsrate}$$

Sowohl innerhalb als auch zwischen Protokoll und Kanal basiert die Kommunikation auf Datenstrukturen. Diese werden entsprechend der ISO/OSI Pack-Entpack-Semantik hierarchisch zusammengesetzt und durch das Archi-

tekturmodell geleitet. Abbildung 5.3 stellt die im Rahmen des Beispiels verwendeten Datenstrukturen dar. Dabei kann zwischen protokollspezifischen und standardisierten Datenstrukturen unterschieden werden. Letztere sind in der Abbildung farblich hervorgehoben. Des weiteren ist anzumerken, dass zu Simulationsbeginn eine Verifikation der aktuellen Protokollstapel- und Kanalparametrisierungen stattfindet, da nicht alle Konfigurationen zueinander kompatibel sind.

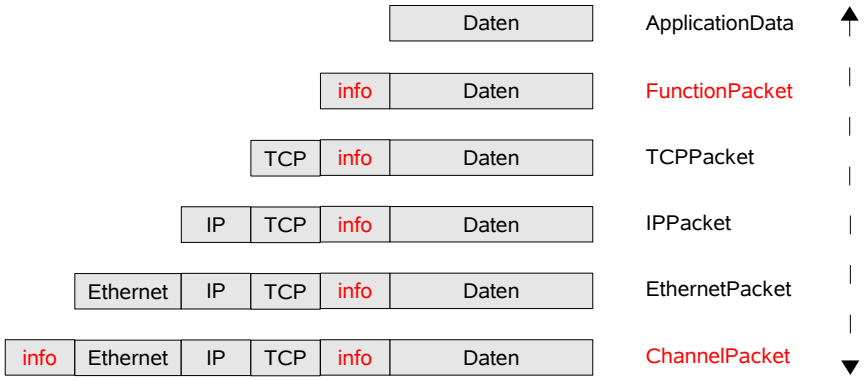

Abbildung 5.3: Pack-Entpack-Semantik des TCP/IP-Ethernet Modells

5.1.4 Validierung

Zur Validierung der entworfenen Architekturkomponenten kommt der Versuchsaufbau aus Abbildung 5.4 zum Einsatz. Dahinter verbirgt sich die Idee, in einem gemeinsamen Modell sowohl eine reale als auch eine virtuelle Datenübertragung zwischen zwei Hosts durchzuführen, um anschließend die Übertragungszeiten zu vergleichen. Im oberen Teil der Abbildung wird die reale Verbindung dargestellt. Diese basiert auf dem Socket-Mechanismus und wird durch die Blöcke `PCSocketClient` sowie `PCSocketServer` beschrieben. Dabei wurde der `PCSocketServer` zum besseren Verständnis mit in die Abbildung aufgenommen, wenngleich er eigentlich innerhalb einer zweiten Rechnerumgebung betrieben wird. Im unteren Teil der Abbildung wird die virtuelle Verbindung veranschaulicht, welche aus zwei Protokollstapeln vom Typ `PCStack` sowie dem Kanal `EthernetChannel` besteht. Prinzipiell beschreiben beide Verbindungsarten ein äquivalentes Verhalten, jedoch wurden bei

den Hosts der realen Verbindung die Schichten unterhalb des eingebetteten `ProtocolFunctionInterface`, sprich der Anwendungsschicht, durch einen Socket-Client bzw. Socket-Server ersetzt. Außerdem wurden die Ports `DataIn` und `DataOut` nach außen geleitet, um eine Anbindung an die gemeinsame Testumgebung zu ermöglichen. Diese besteht aus dem Quell-Block `PulseGen` und dem Senken-Block `XMgraph`. Ersterer ist für die Datengenerierung und letzterer für die grafische Auswertung der Sende- bzw. Empfangszeitpunkte zuständig.

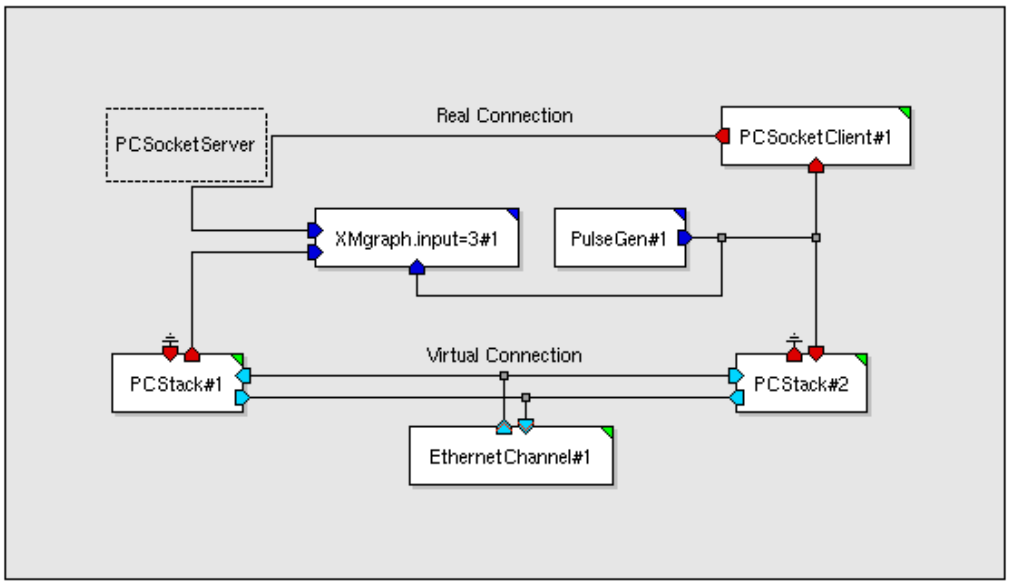

Abbildung 5.4: Validierungsaufbau zum TCP/IP-Ethernet Modell

Bei Ausführung des Modells werden durch den Block `PulseGen` Daten generiert und unmittelbar an den Block `ProtocolFunctionInterface` des realen sowie des virtuellen Hosts gesendet. Daraufhin wird im ersten Fall eine Socket-Verbindung zur Datenübertragung aufgebaut. Im zweiten Fall durchlaufen die Daten beide Protokollstapel und den Kanal. Zur Bemessung der architekturbedingten Verzögerungen wird jeweils vor und nach den Übertragungen die Zeit erfasst. Anzumerken ist, dass bei der realen Übertragung die Ankunftszeit vom Socket-Server protokolliert und zurück an den Socket-Client gesendet wird, um anschließend die Differenz aus Ankunfts- und Sendezeit zu berechnen und die ursprünglichen Daten zu verzögern. Voraussetzung hierfür ist die zeitliche Synchronisation der realen Hosts, was durch den Einsatz des Network Time Protocol (NTP) mit einer Genauigkeit von 0,01 s und besser erreicht wird. Weiterhin ist sicherzustellen, dass alle auf das Netz-

werk zugreifenden Dienste und Programme deaktiviert sind, da ansonsten die Simulationsergebnisse verfälscht werden können. In diesem Zusammenhang ermöglicht das Werkzeug *Tcpdump* eine Analyse und Überwachung des Netzwerkverkehrs. Das Validierungsexperiment wurde anhand folgender Konfiguration durchgeführt:

- Realer Kabeltyp bzw. virtueller Kanal: UTP der Kategorie 5 mit einer Länge von 7,4m
- Reale Hosts: 1,8 GHz Pentium 4 CPU, 512 MB RAM, Betriebssystem OpenSuse 10.2
- Reale Netzwerkkarten bzw. virtuelle Protokollstapel: Standard Ethernet Implementation 100Base-TX (Voll-Duplex, 100 MBit/s), Sende- und Empfangspuffer von 1.000.000 Byte
- Zu übertragende Daten: jeweils 1.000.000 Byte zu den Simulationszeiten 0, 1 und 2

Simulationszeit	**Reale Verzögerung**	**Virtuelle Verzögerung**
0 s	0,0850321 s	0,0950944 s
1 s	0,08502 s	0,09509 s
2 s	0,08502 s	0,09509 s

Tabelle 5.2: Wertevergleich zur Validierung des TCP/IP-Ethernet Modells

Tabelle 5.2 stellt die Ergebnisse des Experiments dar. Die Werte der realen Verbindung entsprechen nahezu dem erwarteten Wert von 0,0831 s, welcher sich wie folgt herleiten lässt. Das maximal übertragbare Datenvolumen eines Ethernet-Pakets beträgt 1500 Byte. Davon ist der Protokoll-Overhead von 20 Byte für IP und weiteren 20 Byte für TCP zu subtrahieren. Hieraus ergibt sich eine maximale Nutzdatengröße von 1460 Byte, was bedeutet, dass 1.000.000 Byte in 685 Paketen übertragen werden. Jedes Paket benötigt wiederum 18 Byte zur Kommunikation über Ethernet. Folglich wächst der Protokoll-Overhead auf insgesamt 58 Byte. Bezogen auf die Paketanzahl, macht das zusammen 1.039.730 Byte bzw. 8.317.840 Bit. Bei einer Übertragungsrate von 100 MBit/s ergibt sich somit eine reine Übertragungszeit von 0,0831 s. Die leichte Diskrepanz gegenüber den ermittelten realen Verzögerungswerten lässt sich auf die unberücksichtigten Protokollverarbeitungszeiten und TCP-Mechanismen zurückführen. Vergleicht man weiterhin die

realen und virtuellen Verzögerungswerte, ist ebenfalls eine Diskrepanz zu erkennen. Ursache hierfür sind die im Modell erfassten Protokollverarbeitungszeiten, welche aufgrund der Abhängigkeit zur jeweiligen Protokollimplementation in Hard- und Software nur schwer bestimmbar sind.

5.2 High Precision Positioning System

5.2.1 Einführung

Im Rahmen des Beispiels wird ein System zur Positionsbestimmung und -ausgabe entwickelt [Baumann u. a. 2007a]. Dabei wird ausgehend von den operationalen Anforderungen die Funktion und Umgebung abgeleitet und zur simulativen Validierung in ein funktionales Gesamtsystemmodell überführt. Daraufhin wird ein architekturelles Gesamtsystemmodell generiert und validiert. Abschließend wird auf Basis des architekturellen Gesamtsystemmodells und implementationsspezifischer Annotierungen Code synthetisiert und durch Ausführung validiert.

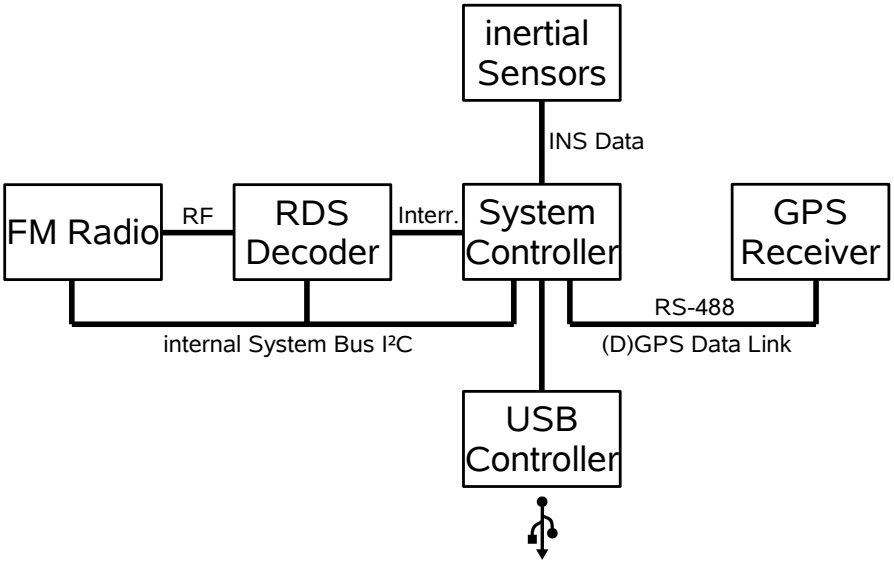

Abbildung 5.5: Konzeptentwurf des High Precision Positioning Systems

Abbildung 5.5 zeigt den im weiteren Verlauf verfolgten Konzeptentwurf. Unter anderem sind mehrere Komponenten zur Positionsbestimmung, eine zen-

trale Steuereinheit zur Verarbeitung der Signale, sowie eine externe USB-Schnittstelle zur Datenentnahme zu erkennen. Dabei beruht die Positionsbestimmung auf einem *Inertial Navigation System* (INS) mit Sensoren für Beschleunigung und Winkelgeschwindigkeit, einem Empfänger für das Global Positioning System (GPS), sowie einem Radio-Empfänger zur Bereitstellung von Differentialdaten in Bezug auf eine Referenzstation. In diesem Zusammenhang wird die Korrektur der via GPS ermittelten Position durch die einer fixen Referenzstation als *Differential GPS* (DGPS) bezeichnet. Aufgabe des Systems ist es nun, die genannten Messwerte zur exakten Positionsbestimmung miteinander zu kombinieren. Hierbei ist zu beachten, dass durch den Konzeptentwurf bereits eine bevorzugte Architektur festgelegt wurde und demzufolge keine strukturelle Architekturiteration notwendig ist.

5.2.2 Implementation

Der Entwurf beginnt mit der Ableitung und Verfeinerung funktionaler Anforderungen aus den operationalen Anforderungen. Diese werden unmittelbar in Form von Funktions- und Umgebungskomponenten in ein funktionales Gesamtsystemmodell überführt, was eine sofortige Validierung während des Entwurfsprozesses ermöglicht. Das entworfene Modell wird in Abbildung 5.6 dargestellt. Im linken unteren Bereich erkennt man mehrere Komponenten für Konfiguration, Empfang und Dekodierung von Radiosignalen. Grundlage bildet das sogenannte Radio Data System (RDS), über das die Position der Referenzstation versendet wird. Diese wird zyklisch ermittelt und an den im unteren zentralen Teil dargestellten Block `GPS Receiver` übertragen. Dort werden die aktuell sichtbaren Satelliten zur Positionsbestimmung ausgewertet und mit Hilfe der Position der Referenzstation korrigiert. Im Ergebnis liegt die Position in Form eines DGPS-Signals vor. Da im späteren realen System ein GPS-Empfänger vom Typ *Trimble SVeeSix* in Kombination mit dem Trimble Standard Interface Protocol (TSIP) [Trim 1999] zum Einsatz kommt, wird das Protokoll auch im Modell verwendet (siehe Abbildung 5.9). Daraufhin gelangen die Positionsdaten zu dem rechts davon dargestellten Block `Position Estimator`, welcher gleichzeitig von den darüberliegenden Blöcken die Positionsdaten des INS empfängt. Insgesamt erfordert das INS zur Positionsbestimmung sechs Messungen (Beschleunigung und Winkelgeschwindigkeit für die drei Raumrichtungen), wobei der Positionsfehler infolge der notwendigen Zweifachintegration mit der Zeit quadratisch anwächst. Zur Kombination der Position des INS und der des DGPS besitzt der Block `Position Estimator` einen Kalman-Filter [Qi u. Moore 2002]. Die dabei ermittelte Position wird letztlich über den unten rechts dargestellten Block

`Results` ausgegeben.

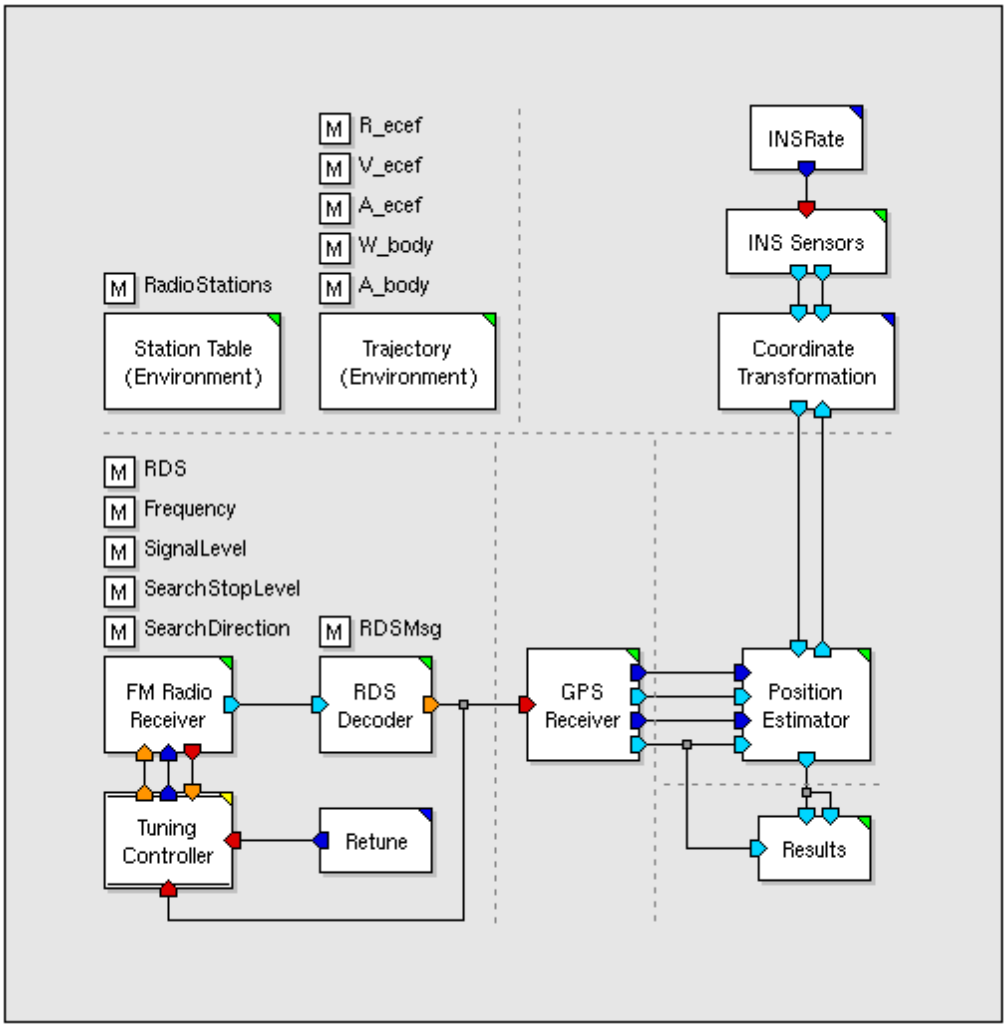

Abbildung 5.6: Funktionales Gesamtsystemmodell des High Precision Positioning Systems

Die Validierung erfolgt anhand der Blöcke `Station Table` und `Trajectory`, durch welche das Umgebungsmodell inklusive Missionen beschrieben wird. Diese werden im oberen linken Teil von Abbildung 5.6 dargestellt. Des weiteren sind mehrere systemweit verfügbare Memory-Komponenten zu erkennen, die gemeinsam den aktuellen Zustand des Umgebungsmodells festhalten und zur Simulationszeit durch die Sensoren des Funktionsmodells ausgelesen werden. Die Missionen unterscheiden sich hierbei durch die Flugbahn des zu ana-

lysieren Körpers, welche entweder emuliert oder via Hardware in the Loop eingelesen werden kann, sichtbare Satelliten sowie verfügbare Radiosender. Eine zu Testzwecken gewählte Flugbahn, sowie das dazugehörige Ergebnis der Positionsbestimmung wird im oberen Teil von Abbildung 5.7 gezeigt. Dabei tritt zunächst eine relativ große Abweichung auf, welche aus dem INS resultiert. Nach einigen Sekunden konvergiert diese jedoch durch die Filterung und verbessert die Positionsbestimmung. Im linken unteren Teil der

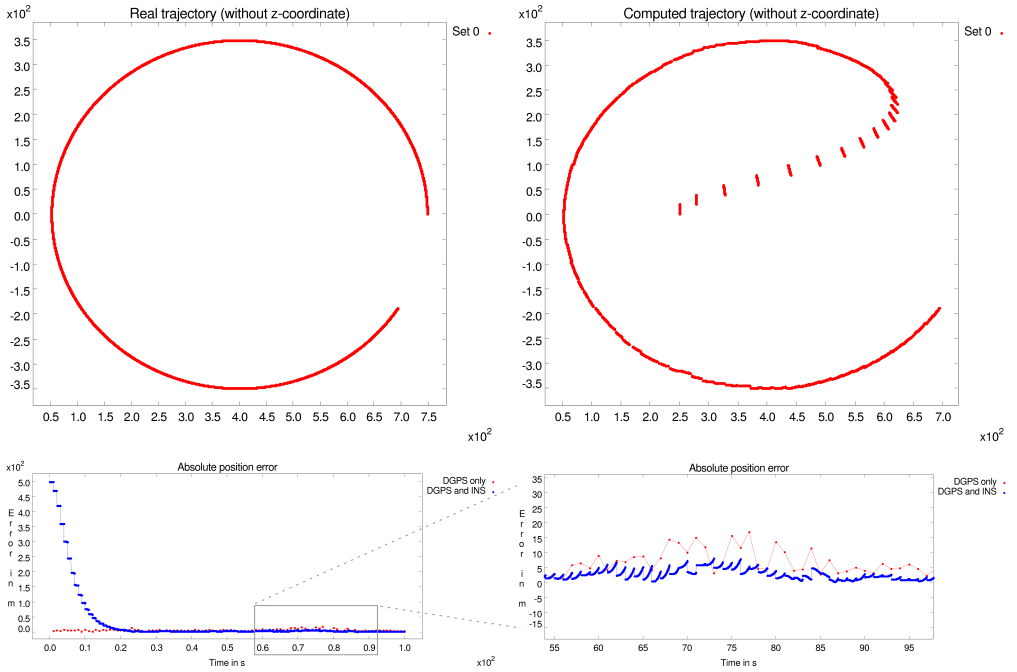

Abbildung 5.7: Trajektorie und Postitionsfehler des High Precision Positioning Systems

Abbildung wird dieser Sachverhalt durch Gegenüberstellung der Positionsfehler bei alleiniger Verwendung von DGPS, sowie in Kombination mit INS genauer betrachtet. Man erkennt, dass sich der Positionsfehler im Rahmen von 3 - 5 m bewegt und bei Hinzunahme von INS weniger stark ausschlägt, was einer Verringerung der Varianz entspricht. Des weiteren wird rechts unten ein Ausschnitt des Positionsfehlers vergrößert dargestellt. Dieser veranschaulicht wie ein kurzzeitiger Ausfall des Radiosenders vom Zeitpunkt 61 bis 81 zu einem Anstieg des Positionsfehlers führt und gleichzeitig durch das INS gedämpft wird. Anzumerken ist, dass das Modell unter Verwendung unterschiedlicher Ausführungsmodelle entworfen wurde, hauptsächlich der DE-,

SDF- und FSM-Domäne. Dabei wurde die Funktionalität in Form von mehreren, über SDF interagierenden, FSM's realisiert.

Zur Überführung des funktionalen Gesamtsystemmodells in ein architekturelles Gesamtsystemmodell müssen zunächst die benötigten Architekturkomponenten entworfen und validiert werden. Dabei ist zu beachten, dass die standardisierten Schnittstellen des Architecture Block Sets einzuhalten sind, da nur so eine automatisierte Integration mit Hilfe des Architektur-Generators möglich ist. Im folgenden werden die verwendeten Protokollspezifikationen sowie deren Umsetzung erläutert:

- Beim *I2C-Bus* handelt es sich um einen seriellen synchronen Zweidraht-Bus zur Verbindungen integrierter Schaltkreise mit einer Maximalgeschwindigkeit 3,4 MBit/s [NXP 2007]. Er besteht aus einer Takt- sowie Datenleitung, besitzt eine Master/Slave-Architektur inklusive Übertragungsprotokoll und unterstützt eine Software-Adressierung. Der im Beispiel entworfene Protokollstapel und Kanal berücksichtigt lediglich die idealisierten Datenübertragungseigenschaften. Dabei werden zur Initialisierung 10 Bits (1 Bit Startbedindung, 7 Bit Empfängeradresse, 1 Bit Festlegung des Lese/Schreibzugriffs, 1 Bit Empfangsbestätigung), während der Datenübertragung für jede 8 zu übertragenden Bits 9 Bits (8 Bit Daten, 1 Bit Empfangsbestätigung) und am Ende 1 Bit als Stoppbedingung benötigt.

- Als *RS-488* wird die vierdrahtige Voll-Duplex-Variante des zweidrahtigen Halb-Duplex-Busses RS-485 bezeichnet, bei welchem es sich wiederum um eine Erweiterung des ebenfalls zweidrahtigen, jedoch unidirektionalen, RS-422 handelt. RS-485 ist eine serielle, differentielle Protokollspezifikation mit einer Master/Slave-Architektur und einer Maximalgeschwindigkeit von 10 MBit/s [Soltero u. a. 2002]. Dabei wird zur Fehlerauffindung das Datensignal über einen Draht invertiert und über den anderen nichtinvertiert übertragen. Analog erfolgt dies bei RS-488 in beide Richtungen. Da kein festes Übertragungsprotokoll existiert, muss dieses durch den Nutzer festgelegt werden. Der im Beispiel entworfene Protokollstapel und Kanal abstrahiert hierbei lediglich die Datenübertragungseigenschaften.

Zusätzlich sei an dieser Stelle die Protokollspezifikation Wishbone erwähnt, welche zur Verbindung der partitionsinternen IP-Cores verwendet wird. Um die erläuterten Architekturkomponenten zu integrieren, muss das funktionale Gesamtsystemmodell durch Annotierungen erweitert werden. Ziel ist es, alle

Funktionsblöcke Partitionen zuzuordnen, sowie die Ausführungsarchitektur jeder Partition und die Kommunikationsarchitektur zwischen den Partitionen festzulegen. Die Strichlinien in Abbildung 5.6 deuten die gewählte Aufteilung bereits an. Daraufhin kann der Architektur-Generator eingesetzt werden, um aus dem annotierten funktionalen Gesamtsystemmodell ein architekturelles Gesamtsystemmodell zu erzeugen. Das Ergebnis wird in Abbildung 5.8 dargestellt. Man erkennt die festgelegten Partitionen, sowie die topologische Struktur, realisiert durch die zuvor entworfenen Kanäle. Unter anderem wurden die Partitionen `System Controller`, `DGPS Radio Receiver` und `USB Controller` über einen gemeinsamen I2C-Bus und die Partitionen `System Controller` und `GPS Receiver` über einen RS-488-Bus verbunden, wobei bei beiden der `System Controller` als Master fungiert. Grund für die Auswahl des RS-488-Busses ist die Tatsache, dass der GPS-Empfänger vom Typ *Trimble SVeeSix* sowohl RS-232 als auch und RS-422 unterstützt. Kanäle vom Typ `Generic` stellen entweder unspezifizierte oder direkte Kommunikationsverbindungen dar.

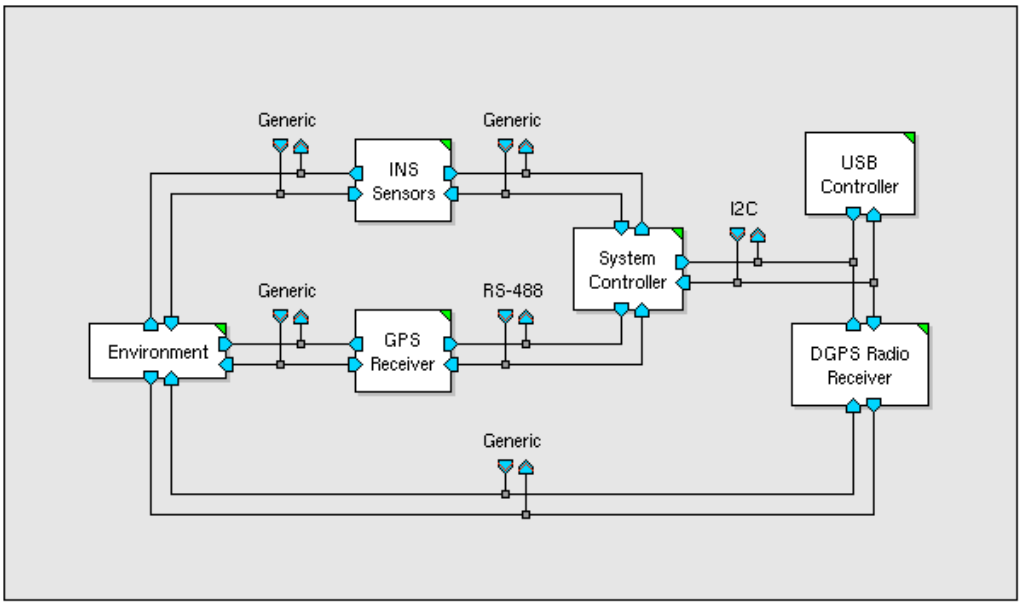

Abbildung 5.8: Architekturelles Gesamtsystemmodell des High Precision Positioning Systems

Um die gewählte Architektur zu validieren, muss das architekturelle Gesamtsystemmodell gegen die nichtfunktionalen Anforderungen verifiziert werden. Dabei ist zu beachten, dass die zuvor verifizierten funktionalen Anforderungen weiterhin erfüllt bleiben. Die Überprüfung wird anhand einer simulativen

Performance-Analyse durch Bemessung der ressourcenabhängigen Verzögerung und Fragmentierung des Datenflusses erreicht. Sind die Verzögerungen zu groß, kann es zu Fehlern oder Ausfällen kommen. In diesem Zusammenhang wurden die Auslastungen der Kommunikationsverbindungen analysiert. Für den I2C-Bus mit einem maximalen Durchsatz von 400 kBit/s konnte eine durchschnittliche Auslastung von circa 5,5908 % ermittelt werden. Die Partition `System Controller` wurde hierbei so konfiguriert, dass sie mit einer Frequenz von 100 Hz Daten von der Partition `DGPS Radio Receiver` anfordert, worauf diese 7 Byte versendet (siehe [Phil 2002]). Zudem sendet die Partition `System Controller` mit der gleichen Frequenz die Positionsdaten in Form der geografischen Breite und Länge an die Partition `USB Controller`. Diese bestehen insgesamt aus 16 Byte, jeweils 2 Byte für den Grad, 2 Byte für die Minute und 4 Byte für die Dezimalsekunde. Für den RS-488-Bus mit einem maximalen Durchsatz von 9600 Bit/s konnte bei einer Abfragefrequenz von 10 Hz in Richtung `GPS Receiver` eine Auslastung von 4,5833 % und entgegengesetzt eine Auslastung von 55,0000 % gemessen werden. Dabei kam ein serielles Standardprotokoll mit 1 Start-Bit, 8 Daten-Bits, 1 Paritäts-Bit und 1 Stopp-Bit zum Einsatz. Zudem wurde zur Konfiguration und Steuerung des Datenaustausches TSIP verwendet. Dieses unterscheidet zwischen einem `Command Packet` zur Steuerung des GPS-Empfängers und einem `Report Packet` für dessen Antworten. Beide besitzen eine feste Grundstruktur, bestehend aus 1 Start-Byte, 1 Identifikations-Byte, den zu übertragenden Daten-Bytes und 2 End-Bytes. Daraus folgt, dass für jedes Paket 4 Byte Protokoll-Overhead benötigt werden. Bei Systemausführung kommunizieren die Partitionen über das Command Packet `Last Position and Velocity Request` (0 Daten-Byte) zur Anforderung von Position und Geschwindigkeit, sowie über die Report Packets `Last Computed Fix Report` (8 Daten-Byte), `Single-Precision Position Fix (XYZ Cartesian ECEF) Report` (16 Daten-Byte) und `Velocity Fix (XYZ Cartesian ECEF) Report` (20 Daten-Byte) als Reaktion auf die Anforderung. Im Ergebnis der Analyse konnte gezeigt werden, dass die Systemfunktionalität mit der gewählten Architektur realisierbar ist.

Auf Grundlage des architekturellen Gesamtsystemmodells kann nun die eigentliche Implementation synthetisiert werden. Da infolge des erhöhten Abstraktionsniveaus implementationsspezifische Details fehlen, sind diese durch den Nutzer zu ergänzen, das heißt in Form von Annotierungen den Modellelementen hinzuzufügen. Zur Festlegung bietet MLDesigner für jedes Modellelement einen Parameter namens `Annotation` an, welcher bei Modellspeicherung in das XML-Attribut `annotation` überführt wird. Zudem wurde eine entsprechende Annotierungs-Sprache entwickelt. Im Anschluss kann

die Transformation des annotierten Modells in Hard- und Software gestartet werden. Dabei wird die XML-Repräsentation der Modelle auf dem Dateisystem anhand von XSLT-Skripten ausgelesen und in die entsprechende Sprache transformiert. Die Komponenten des architekturellen Gesamtsystemmodells werden wie folgt behandelt:

- Funktionen von Partitionen werden zu IP-Cores oder Prozessen
- Protokollstapel der Partitionen werden zu IP-Cores aus einer wiederverwendbaren, auf Wishbone basierenden Bibliothek
- Kanäle werden zu Verbindungen zwischen IP-Cores auf Grundlage von Wishbone oder bei externen Verbindungen auf Grundlage anderweitiger Übertragungsmedien

Durch Entwurf neuer oder Modifikation vorhandener XSLT-Skripte können beliebige Beschreibungssprachen synthetisiert werden. Aktuell existieren die im Rahmen den Beispiels verwendeten VHDL-Transformationsvorschriften bezüglich Field Programmable Gate Arrays (FPGA) und Application Specific Integrated Circuits (ASIC). Anzumerken ist, dass ein iterativer Verfeinerungs- und Syntheseprozess unterstützt wird, bei dem direkte Modifikationen am synthetisierten Code nicht verloren gehen.

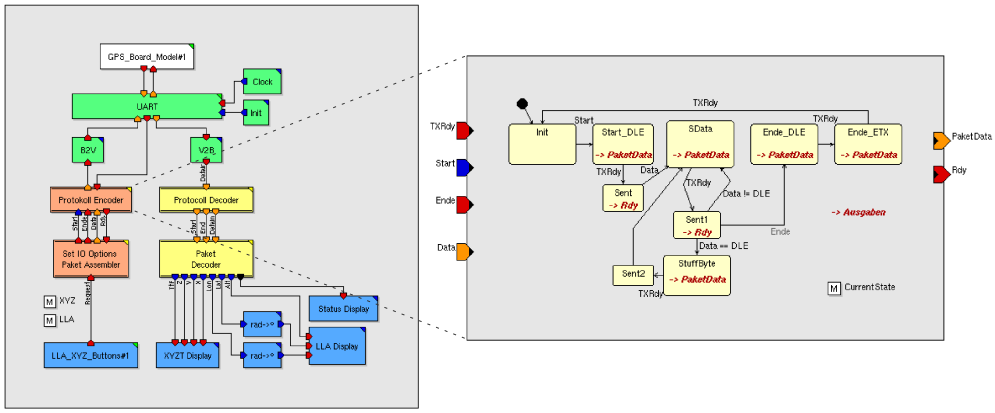

Abbildung 5.9: Submodell Protocol Encoder des GPS Receivers

Im weiteren Verlauf wird die Synthese am Beispiel des Blocks `Protocol Encoder` erläutert. Dieser ist Bestandteil des Blocks `GPS Receiver`, welcher

unter anderem die Codierung von Steuerungssignalen und die Decodierung von GPS-Daten realisiert. Abbildung 5.9 stellt die zugehörigen Modelle dar. Im linken Teil erkennt man die unterschiedlichen Submodelle des Blocks `GPS Receiver` und im rechten die Verhaltensbeschreibung des Blocks `Protocol Encoder` in Form einer FSM bzw. Zustandsmaschine. Des weiteren ist zu erkennen, dass der Block `Protocol Encoder` mehrere Ports besitzt. Diese müssen um Annotierungen bezüglich Busart, Busbreite, Zugriffsart, und Granularität erweitert werden. Tabelle 5.3 listet die Ports, sowie die zugehörigen Annotierungen auf. Hierbei wird bei der Busart zwischen zwischen `[WB]` und `[EXTERN]` unterschieden. Ersteres steht für interne Wishbone-Kanäle zur Verbindung von IP-Cores. Letzteres für externe, die Partition verlassende Kanäle. Des weiteren sind in der Abbildung mehrere Zustände zu erkennen, welche jedoch direkt, sprich ohne zusätzliche Annotierungen, überführt werden können [Rath u. Salzwedel 2004]. Basierend auf der XML-Beschreibung des Blocks `Protocol Encoder` kann anschließend anhand von XSLT-Skripten ein IP-Core in Form einer VHDL-Beschreibung erzeugt werden (siehe Anhang D). Diese besteht aus der Entität `ProtocolEncoder` zur Definition der externen Schnittstelle, sowie der Architektur `ProtocolEncoder1` zur Definition von Struktur und Verhalten.

Portname	**Annotierung**
Start	[WB] access=w; width=1;
End	[WB] width=1; access=w;
Data	[WB] width=8; granularity=8; access=w;
PaketData	[EXTERN] width=8; granularity=8; access=r;
TXRdy	[EXTERN] width=1;
Rdy	[WB] width=1;

Tabelle 5.3: Annotierungen am Beispiel des Protocol Encoders

Analog dazu werden auch die anderen Submodelle des Blocks `GPS Receiver` transformiert. Im Ergebnis liegt ein Netzwerk kommunizierender IP-Cores vor, welche unter Verwendung von Quartus [Alte 2005] veranschaulicht werden können. Quartus ist ein modellbasiertes Werkzeug für den Entwurf von Programmable Logic Devices (PLD) und unterstützt unter anderem VHDL-Beschreibungen. Es besitzt eine grafische Benutzeröberfläche zur Darstellung sowie Modifikation logischer Schaltkreise, Simulationsfähigkeiten und erlaubt die Überführung von IP-Cores auf ECU's. Abbildung 5.10 zeigt die zuvor synthetisierten IP-Cores in Quartus. Man erkennt, dass diese den Submodellen des Blocks `GPS Receiver` entsprechen und untereinander auf Grundlage von Wishbone kommunizieren. Ausgehend vom Quartus-Modell oder den realen

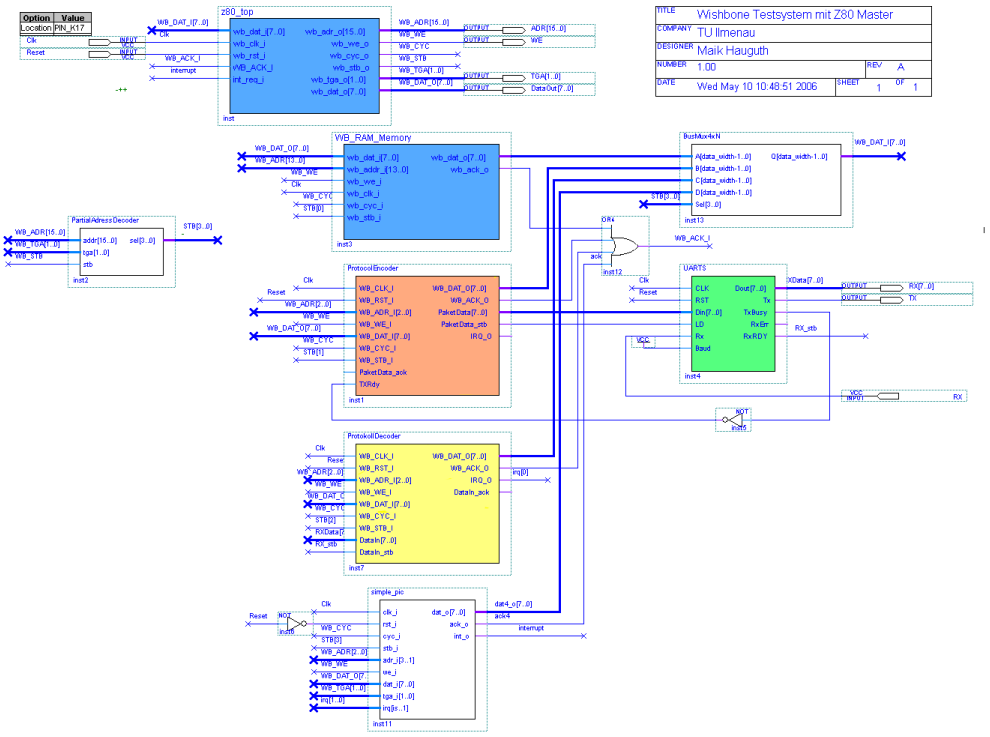

Abbildung 5.10: Vernetzte IP-Cores des GPS Receivers

ECU's muss abschließend eine Validierung durchgeführt werden. Dabei wird das System ausgeführt und gegen die zuvor definierten Anforderungen und Missionen getestet.

5.2.3 Bewertung

Das Beispiel zeigte die Anwendung der Prozessschritte des erweiterten Mission Level Designs sowie deren Beziehungen untereinander (siehe Abbildung 3.2). Dabei wurde während der funktionalen Entwurfsphase unter anderem ein Navigationsfilter modellbasiert entworfen und simulativ gegen die funktionalen Anforderungen getestet. Anschließend kam ein Modellgenerator zum Einsatz, welcher mehrere zuvor entworfene Architekturkomponenten automatisiert in das Modell integrierte. Dies ermöglichte die Durchführung einer simulativen Performance-Analyse zur gemeinsamen Überprüfung der funktio-

nalen und nichtfunktionalen Anforderungen. Im Ergebnis konnte die gewählte Architektur und deren Parametrisierung bestätigt werden. Daraufhin wurde das Modell annotiert und anhand von Transformationsvorschriften in eine reale Implementation überführt.

Zusammenfassend konnte gezeigt werden, dass die Entscheidungsfindung, wie die Festlegung der Funktion und die Wahl einer geeigneten Architektur, in die frühen Entwurfsphasen verlegt werden kann und die Überführung in eine reale Implementation erst abschließend notwendig ist. Zudem wurde gezeigt, dass ein abstraktes, ausführbares Modell durch Hinzufügen plattformspezifischer Zusatzinformationen in eine reale Implementation transformiert werden kann, wobei die Unabhängig des Modells von der Implementation erhalten bleibt. In diesem Zusammenhang bildeten die zuvor festgelegten Partitionen der standardisierten Architekturbeschreibung den Ausgangspunkt. Auch stellte sich die Verwendung von Modellbeschreibungen in XML und Modelltransformationen in XSLT als eine relativ schnell anpassbare und erweiterbare Plattform zur Synthetisierung heraus.

5.3 Avionik-Architekturen

5.3.1 Einführung

Der im weiteren Verlauf erläuterte Entwurfsprozess beschäftigt sich mit der Anwendung des Architecture Block Sets zur Planung und Optimierung von Avionik-Architekturen. Dabei werden ausgehend von der funktionalen Beschreibung der Flugzeugsysteme verschiedene Architektur-Varianten abgeleitet [Fischer 2007]. Des weiteren kommt zur Partitionierung, also der Zuordnung von Funktion auf Architektur, ein Ressourcenmodell zum Einsatz. Im folgenden werden die drei Phasen des Entwurfsprozesses erläutert.

Aufgabe der ersten Phase ist es, alle notwendigen Vorgaben und Anforderungen der zu integrierenden Flugzeugsysteme und des gesamten Flugzeugs zu erfassen. Abbildung 5.11 stellt die wichtigsten Punkte im Überblick dar. Darauf basierend werden architekturunabhängige, ausführbare Funktionsmodelle erstellt und validiert. Diese beinhalten sämtliche Flugzeugsysteme und werden nach Air Transport Association (ATA) Kategorien gruppiert. Hierbei wird unter einer ATA ein international gültiger Bezeichner für ein Flugzeugsystem bzw. Subsystem in Form eines Zahlencodes verstanden.

In der zweiten Phase erfolgt die Zuordnung aller Systemkomponenten bzw.

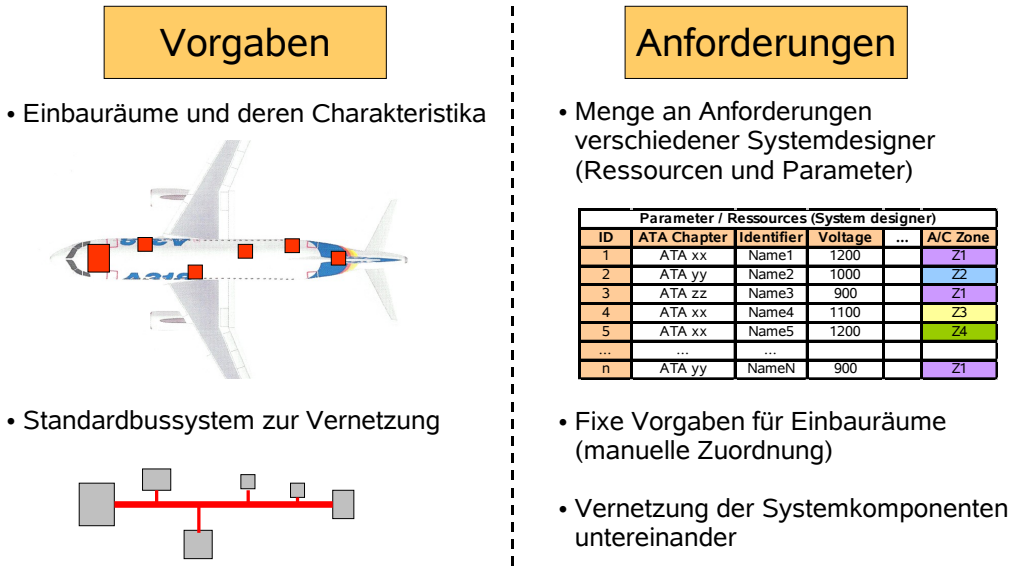

Abbildung 5.11: Vorgaben und Anforderungen an die Avionik-Architektur [Fischer 2007]

Subsysteme auf eine topologische Avionik-Architektur, sowie die Determinierung der Kanäle zwischen den Einbauräumen. Hierzu werden die Line Replaceable Module (LRM) und Line Replaceable Unit (LRU) Komponenten der verschiedenen Flugzeugsysteme durch Simulation eines speziellen Ressourcenmodells im Flugzeug verteilt. Viele der Steuergeräte bzw. LRU's können frei im Flugzeug platziert werden, während andere Geräte, wie beispielsweise Sensoren oder Motoren, einem fest vorgegebenen Platz zugeordnet werden. Dabei entspricht die Zuordnung prinzipiell dem Partitionierungsprozess aus Abschnitt 3.3. Im Ergebnis der Simulation liegt eine XML-basierte Konfigurationsdatei als Eingabe für die dritte Phase vor. Abbildung 5.12 zeigt eine ermittelte Verteilungskonfiguration am Beispiel. Einbauräume werden rot und die Komponenten des Flugzeugsystems grün dargestellt. Der vorgegebene Ort ist dabei nichts anderes als eine Anforderung, die bei der Verteilung der Komponenten zu beachten ist. Anzumerken ist, das während des Partitionierungsprozesses Methoden zur Optimierung der Avionik-Architektur angewendet werden. Diese beeinflussen die Verteilung hinsichtlich verschiedener Gesichtspunkte, wie beispielsweise der Einsparung von Kabeln, Gewicht und Kosten.

Die dritte Phase ist durch die Erzeugung eines ausführbaren Architektur-

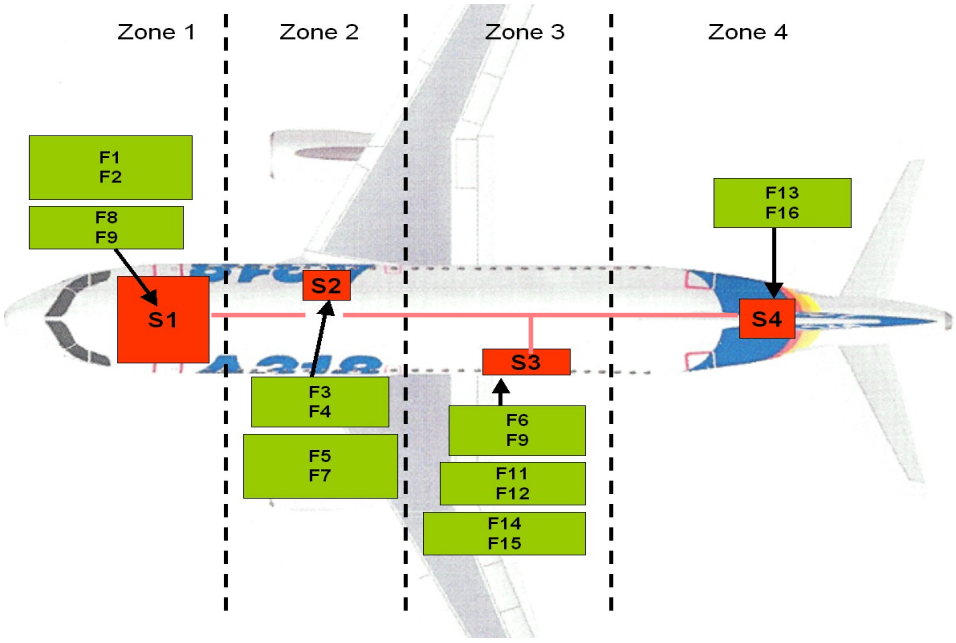

Abbildung 5.12: Verteilung auf die Avionik-Architektur [Fischer 2007]

modells bzw. architekturellen Gesamtsystemmodells gekennzeichnet. Dabei dienen die Funktionsmodelle sowie die Konfigurationsdatei aus den vorhergehenden Phasen als Ausgangspunkt. Zur Realisierung kommt das Architecture Block Set mit seinem Architektur-Generator und den bereits existierenden Architekturkomponenten zum Einsatz.

Um unterschiedliche Architekturmodelle zu erzeugen und miteinander zu vergleichen, wird die zweite Phase des Entwurfsprozesses beliebig oft wiederholt. Hierdurch können ausgehend von den einmal definierten und validierten Funktionsmodellen beliebig viele Architekturmodelle abgeleitet werden. Für ein Flugzeug ist es somit möglich aus einer Menge alternativer Avionik-Architekturen zu wählen und diese auf Grundlage verschiedener Optimierkriterien zu erzeugen. Anzumerken ist, dass für einen korrekten Ablauf der Phasen, die Funktionsmodelle sowie das Ressourcenmodell stets zueinander konsistent sein müssen.

5.3.2 Implementation

Die Verwendung des Entwurfsprozesses wird im folgenden am Beispiel eines Systems zur Regulierung des Kabinendrucks erläutert. Dabei handelt es sich um eine vereinfachte Beschreibung des Pressure Control and Monitoring Systems, entsprechend ATA 21-31. Jedes Flugzeug mit einem Einsatzbereich von über 10000 Fuß benötigt eine Druckkabine. Durch diese wird einerseits eine überlebensfähige sowie angenehme Umwelt für die Passagiere geschaffen und andererseits die strukturelle Integrität des Flugzeugs sichergestellt. Während des normalen Flugbetriebs strömt ein kontinuierlicher Luftstrom in die Kabine, welcher mit Hilfe von Ventilen, den sogenannten Outflow Valves (OFV), gesteuert wird. Je nach Höhe regulieren die OFV die Menge des Luftabflus-

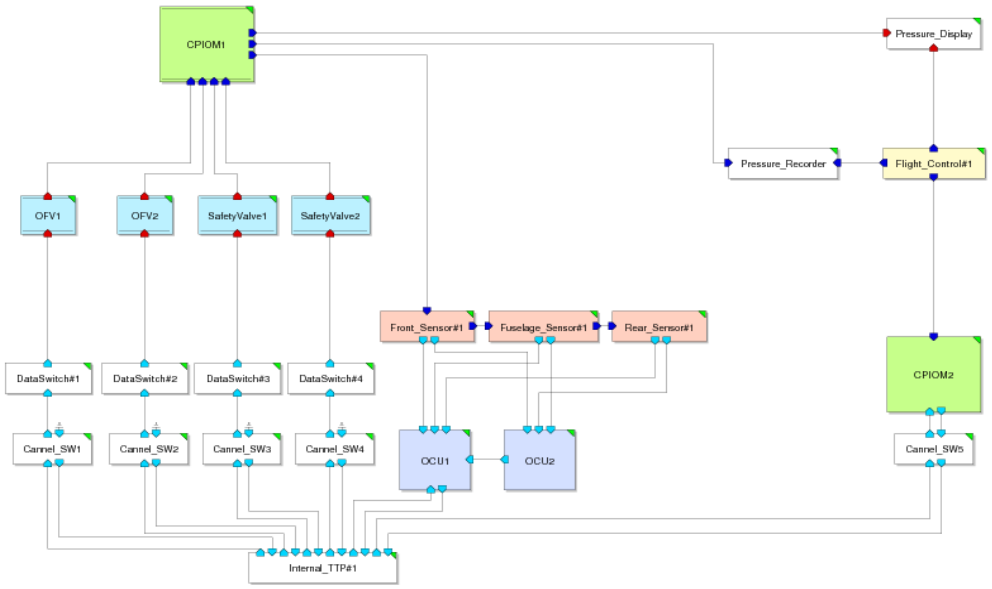

Abbildung 5.13: Funktionsmodell zur Avionik-Architektur [Fischer 2007]

ses aus der Kabine, so dass der bei 8000 Fuß vorherrschende Innendruck nie überschritten wird. Zur korrekten Bestimmung des Innendrucks sind mehrere Sensoren im gesamten Flugzeug verteilt. Zusätzlich zu den OFV existieren noch weitere Ventile. Die Savety Valves (SV) übernehmen im Notfall die Aufgaben der OFV, die Overpressure Relief Valves (ORV) ermöglichen die Verminderung des Kabinendrucks und die Negative Differential Pressure Relief Valves (NRV) stellen im Notfall einen minimalen Druck sicher. Zur

Steuerung des gesamten Systems kommen Core Processing Input Output Modules (CPIOM), bei welchen es sich um konfigurierbare Ressourcen für beliebige Services handelt, sowie als Outflow Valve Control and Sensor Units (OCU) bezeichnete Steuergeräte zum Einsatz. Abbildung 5.13 stellt das in MLDesigner entworfene Funktionsmodell dar.

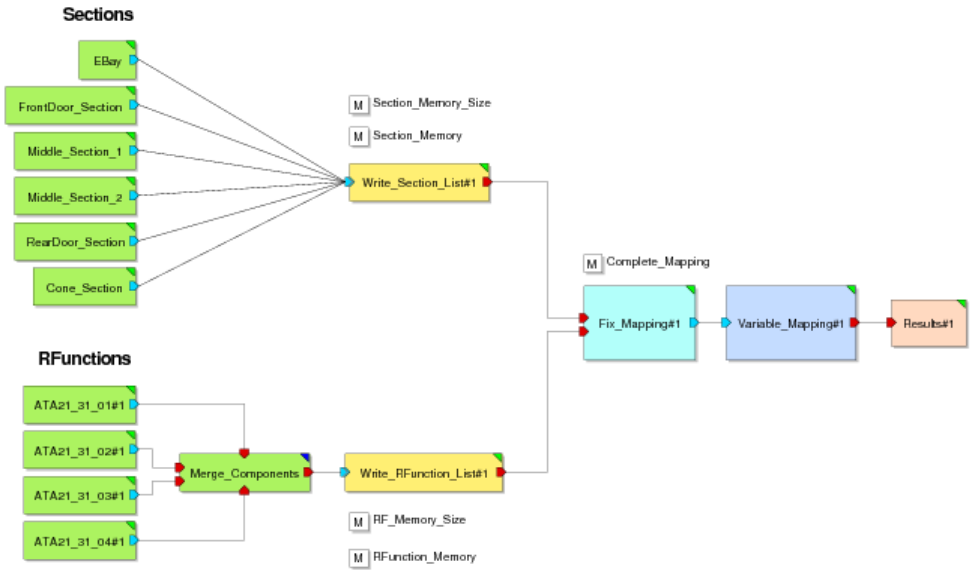

Abbildung 5.14: Ressourcenmodell zur Avionik-Architektur [Fischer 2007]

Parallel zum Funktionsmodell existiert ein abstraktes Ressourcenmodell zur Ermittlung und Optimierung der Avionik-Architektur. Dieses setzt die zuvor in Bezug auf das Flugzeug und seine Systeme getroffenen Vorgaben und Anforderungen um. Hierbei dient die Spezifikation eines Airbus A320 als Ausgangspunkt. Durch Ausführung des Ressourcenmodells werden alle RFunctions auf Sections, respektive Funktionen auf Partitionen, verteilt. Im Ergebnis wurde hinsichtlich einer Referenz-Architektur eine Kabeleinsparung von 68 %, ein verringertes Gewicht, sowie eine erhöhte Verfügbarkeit erreicht. Als weiteres Ergebnis liegt eine XML-Datei vor, welche die ermittelte Belegung bzw. Partitionierung beinhaltet und als Eingabe für den Architektur-Generator dient. Das entsprechende System wird in Abbildung 5.14 dargestellt und besteht aus folgenden Komponenten:

- `RFunction`-Module setzen funktionale und `Section`-Module architekturelle Vorgaben und Anforderungen mittels Parameterwerten um

- Module vom Typ `Write_RFunction_List` legen die Parameterwerte der RFunctions im globalen Memory `RFunction_Memory` ab und Module vom Typ `Write_Section_List` die der Sections im globalen Memory `Section_Memory`
- Das Modul `Fix_Mapping` ordnet bestimmten RFunctions feste Sections zu und speichert die Belegung im globalen Memory `Complete_Mapping`
- Durch das Modul `Variable_Mapping` wird die variable Zuordnung von RFunctions auf Sections mittels Optimierungsalgorithmen vollzogen und die Belegung im globalen Memory `Complete_Mapping` gespeichert
- Nach Abschluss der Architekturermittlung sorgt das Modul `Results` für die Anzeige der Ergebnisse und die Erzeugung von Ausgabedateien

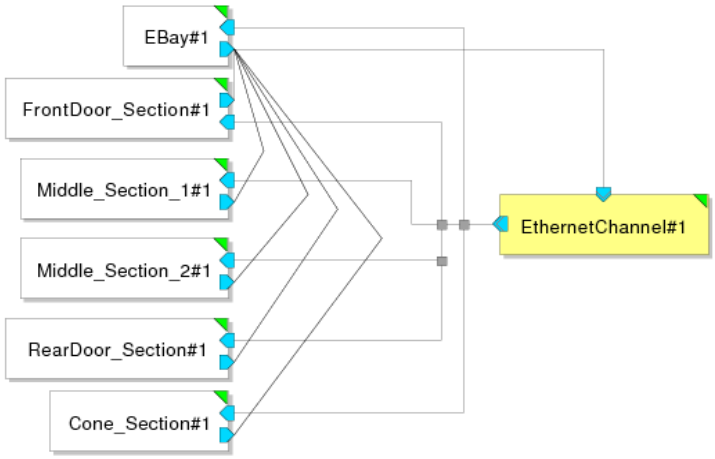

Abbildung 5.15: Architekturmodell zur Avionik-Architektur [Fischer 2007]

Das im Anschluss generierte Architekturmodell bzw. architekturelle Gesamtsystemmodell wird in Abbildung 5.15 dargestellt. Es besteht aus mehreren Partitionen und Kanälen, wobei letztere zwischen jeder Partition existieren und lediglich überlagert abgebildet werden. Dabei kommen unter anderem die in Abschnitt 5.1 erstellten Protokollstapel und Kanäle der Protokollfamilie von TCP/IP-Ethernet zum Einsatz. Wird das Architekturmodell ausgeführt, kann das dynamische Verhalten einzelner Funktionskomponenten in Kombination mit Architekturkomponenten betrachtet werden. Demzufolge wird nicht nur eine Analyse und Validierung auf funktionaler Ebene, sondern

ebenfalls auf elektronischer bzw. architektureller Ebene unterstützt. Werden gleich mehrere Funktionsmodelle von Flugzeugsystemen innerhalb eines Architekturmodells zusammengeführt, ist auch eine gemeinsame Betrachtung möglich. Folglich können Interferenzen zwischen einzelnen Funktionsmodellen aufgedeckt werden. Bottlenecks, Jitter und andere Schwachstellen der verwendeten Architektur werden sichtbar. Die ermittelten architekturbedingten

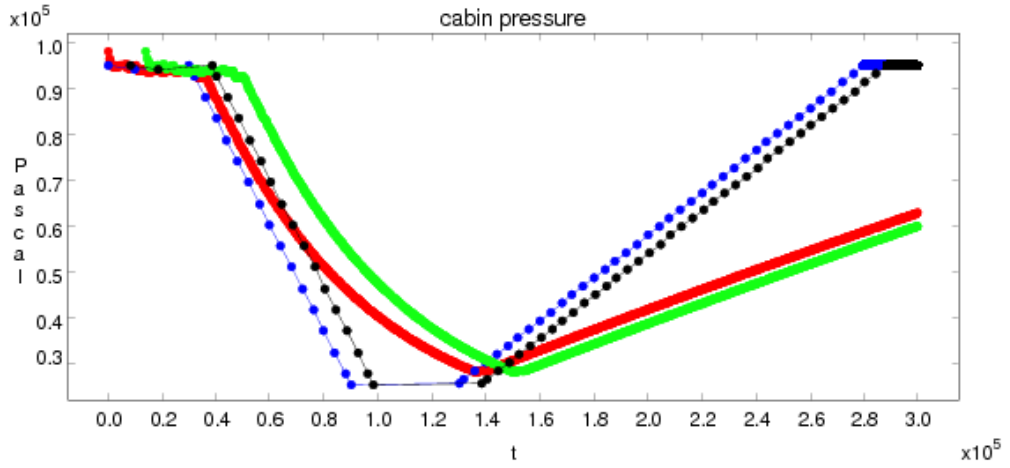

Abbildung 5.16: Verzögerungen durch die Avionik-Architektur [Fischer 2007]

Verzögerungen werden in Abbildung 5.16 in Bezug auf die Druckregulierung dargestellt. Blau steht für den Soll-Druck im Funktionsmodell, rot für den Ist-Druck im Funktionsmodell, schwarz für den Soll-Druck im Architekturmodell und grün für den Ist-Druck im Architekturmodell. Man erkennt deutlich, dass das funktionale Verhalten leicht verzögert auch im Architekturmodell gewährleistet ist.

5.3.3 Bewertung

Der Entwurfsprozess für Avionik-Architekturen beruht auf dem erweiterten Mission Level Design sowie den Komponenten des Architecture Block Sets. Unter anderem wurde der Architektur-Generator in Verbindung mit einer XML-basierten Konfigurationsdatei eingesetzt, um aus einem funktionalen Gesamtsystemmodell mehrere architekturelle Gesamtsystemmodelle abzuleiten. Abbildung 5.17 veranschaulicht den Entwurfsprozess anhand eines Akti-

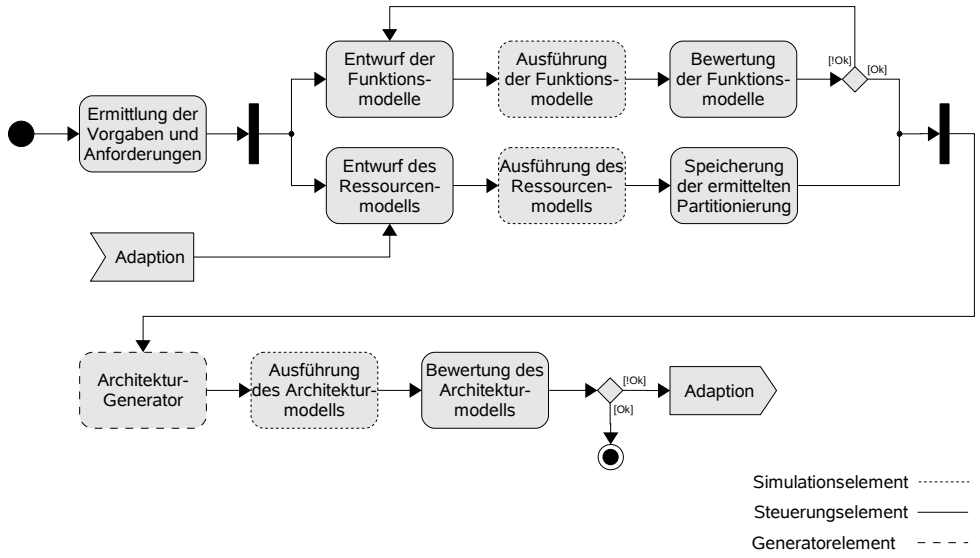

Abbildung 5.17: Entwurfsprozess zur Avionik-Architektur

vitätsdiagramms. Dieses beinhaltet zwei Iterationsschleifen. Die erste bezieht sich auf den Entwurf von Funktionsmodellen, die zweite auf die Ermittlung von Partitionierungen und die Generierung von Architekturmodellen. Des weiteren ist ein Parallelisierungs- sowie ein Synchronisationsknoten zu erkennen, durch welche der Kontrollfluss in zwei gleichzeitig ausführbare Bereiche aufgespalten wird. Bezieht man die verwendeten Aktivitäten auf die Gruppierung aus Abschnitt 3.4, kann auch hier zwischen Simulations-, Steuerungs- und Generatorelementen unterschieden werden. Demnach wird die Zielstellung eines ausführbaren Entwurfsprozesses zur Entwurfsraumanalyse und damit zur weitergehenden Automatisierung verfolgt. Die Realisierung kann beispielsweise durch ein kontrollierendes MLDesigner-System erfolgen.

Kapitel 6

Zusammenfassung

6.1 Ergebnisse

Im Rahmen der Arbeit wurden Methoden zur Steigerung der Spezifikationsqualität und -geschwindigkeit beim Entwurf von Gesamtsystemmodellen in den frühen Entwurfsphasen entwickelt. Dabei diente der Entwurfsansatz Mission Level Design aus Ausgangspunkt. Dieser unterstützt bereits einen integrierten Entwurf auf Gesamtsystemebene unter Berücksichtigung der Kopplungseffekte zwischen Subsystemen. Systemspezifikationen liegen unmittelbar in ausführbarer Form vor und können frühzeitig verifiziert und validiert werden. Da jedoch keine Entwurfsebene zur automatisierten Iteration über Entwurfsalternativen existiert, ist die Auffindung einer möglichen bzw. optimalen Architektur mit einem hohen Entwurfsaufwand verbunden. Um diesen Aufwand zu minimieren, wurde eine Anwendungsumgebung zur automatisierten Erzeugung von Gesamtsystemmodellen mit unterschiedlichen Architekturen, sowie zur vereinheitlichten Leistungsbewertung bzw. Performance-Analyse entwickelt. Anschließend wurde ein Entwurfsprozess zur Iteration über Architekturalternativen in den frühen Entwurfsphasen definiert.

Hierzu musste zunächst eine standardisierte, ausführbare Architekturbeschreibung für Gesamtsystemmodelle entwickelt werden. Nur so ist ein Entwurf austauschbarer Architekturkomponenten mit determinierten Schnittstellen, ein flexibles Mapping von Funktion in Architektur und umgekehrt, sowie eine automatisierte Erzeugung von Gesamtsystemmodellen möglich. Letzteres konnte durch Implementation eines XML-basierten Modellgenerators erreicht werden. Um die standardisierte Architekturbeschreibung und den dazugehörigen Entwurfsprozess für Architekturkomponenten zu validieren, wurden

mehrere Beispielmodelle entworfen und mit realen Systemen verglichen. Unter anderem wurde ein Modell für die Protokollfamilie von TCP/IP-Ethernet entwickelt (siehe 5.1).

Im Anschluss wurden Untersuchungen zur Automatisierung des definierten Entwurfsprozesses durchgeführt. Dabei konnte gezeigt werden, dass dieser in eine ausführbare Spezifikation überführt werden kann, um eine automatisierte Iteration über Architekturalternativen zu ermöglichen. Grundlage bildet ein ausführbares Aktivitätsdiagramm mit interagierenden Simulations-, Steuerungs- und Generatorelementen. In diesem Zusammenhang kam der Begriff eines ausführbaren Entwurfsprozesses zur Entwurfsraumanalyse zum Einsatz. Zur Validierung des Entwurfsprozesses wurde eine Optimierung am Beispiel einer Avionik-Architektur durchgeführt (siehe 5.3).

Zusätzlich zur Architekturoptimierung und der damit verbundenen Modellgenerierung und Leistungsbewertung wurden auf Grundlage der standardisierten Architekturbeschreibung Transformationsvorschriften zur plattformsspezifischen Synthese entwickelt. Diese ermöglichen die Überführung des abstrakten Entwurfs in IP-Cores, unter anderem auch unter Einbindung bereits existierender IP-Cores. Da das Gesamtsystemmodell durch ein Fehlen implementationsspezifischer Details gekennzeichnet ist, musste eine Annotierungs-Sprache zur Verfeinerung der Modellkomponenten entwickelt werden. Hierdurch konnte die XML-Repräsentation des annotierten Gesamtsystemmodells anhand von XSLT-Skripten in vernetzte IP-Cores transformiert werden. Der Partitionierungs- und Syntheseprozess wurde am Beispiel eines Systems zur GPS-basierten Positionsbestimmung validiert (siehe 5.2).

Durch die erläuterten Erweiterungen des Mission Level Designs konnte eine Erhöhung des Automatisierungsgrads während des Entwurfsprozesses erreicht werden. Abbildung 6.1 fast die im Rahmen der Arbeit verfolgte Vorgehensweise in Form eines Aktivitätsdiagramms bezüglich der Zielstellungen aus Abschnitt 1.2 zusammen. Zunächst wurden die Anforderungen an Entwurfsansätze analysiert. Darauf aufbauend konnten die notwendigen Erweiterungen hinsichtlich des Entwurfsansatzes Mission Level Design definiert werden. Schlussendlich wurden die Erweiterungen unter Verwendung von MLDesigner umgesetzt und validiert. Bezieht man die Ergebnisse auf die in Abschnitt 1.1 aufgeführten Entwurfsprobleme, lassen sich folgende Verbesserungen feststellen:

- **System Design Gap:**
  Durch die automatisierte Generierung von Gesamtsystemmodellen mit unterschiedlichen Architekturen und die vereinheitlichte Performance-

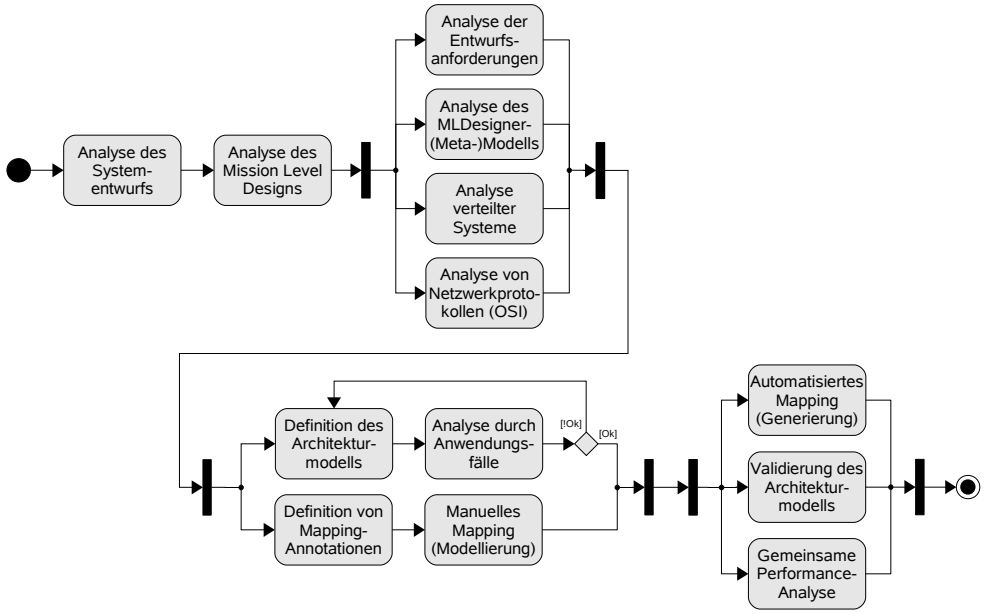

Abbildung 6.1: Zusammenfassung zur Vorgehensweise

Analyse, kann bereits in den frühen Entwurfsphasen eine optimale Architektur ermittelt werden. Dies bedeutet, dass eine frühzeitige Entwurfsraumanalyse zur Entscheidungsfindung möglich ist.

- **System Abstraction Gap:**
  Die Festlegung einer Architektur inklusive Partitionierung wurde in die frühen Entwurfsphasen auf Gesamtsystemebene verlagert. Ausgehend von der dort bestimmten Architektur in Verbindung mit implementationsspezifischen Zusatzinformationen, ist eine automatisierten Generierung von Implementationen möglich. Folglich wird eine Synthese auf Grundlage abstrakter Modelle unterstützt.

- **Design Space Exploration:**
  Der Prozess zur Architekturoptimierung konnte automatisiert werden. Hierzu wurden ausführbare Gesamtsystemmodelle in einen ausführbaren Entwurfsprozess integriert. Dies ermöglicht eine iterative Erzeugung, Ausführung und Bewertung einzelner Entwürfe. In Kombination mit Zielfunktionen und Optimierungsheuristiken kann so der Entwurfsraum automatisiert durchlaufen werden.

6.2 Ausblick

Aus den vorgestellten Erweiterungen des Mission Level Designs und der Umsetzung in Form einer MLDesigner-Anwendungsumgebung ergeben sich eine Reihe von Ansatzpunkten für weiterführende Arbeiten. Folgende Übersicht fasst diese zusammen:

- Durch Entwurf und Validierung zusätzlicher Ausführungs- und Kommunikationskomponenten kann die Architekturbibliothek erweitert werden. Umso größer die Architekturbibliothek ist, desto mehr Auswahlmöglichkeiten existieren beim Architekturentwurf.
- Die Funktionalität des neu entwickelten ESL-Targets muss zur vereinfachten Anwendung direkt in MLDesigner integriert werden. Dies ist zudem Voraussetzung für die Entwicklung eines Verteilungsdiagramms zur Architekturmodellierung inklusive Funktionszuordnung und deren Integration in ein Missionsumfeld. Dabei ist zu beachten, dass das Verteilungsdiagramm zur Ausführung um die Semantik der DE-Domäne zu erweitern ist.
- Zur Vereinfachung der Partitionierung und Modellgenerierung sollten mehrere MLDesigner-Features neu gestaltet bzw. angepasst werden. Zum einen ist ein funktionierender Input-Output-Port einzuführen, um einen bidirektionalen Kommunikationsfluss über genau eine Relation zu ermöglichen. Des weiteren sind die hierarchisch verlinkbaren Argument-Modellelemente auf Kernel-Ebene durch Blöcke abzubilden, damit die Kommunikation zur Ausführungszeit ausschließlich über Ports stattfindet. Auch sollten Synchronisationspunkte, sprich Auto-Forks und Auto-Merges, zur eindeutigen Partitionierung nicht nur auf Kernel-Ebene existieren, sondern auch eine grafische Block-Repräsentation besitzen. Eine weitere Verbesserung kann durch die Offenlegung der interen Modellelement-Identifikatoren erreicht werden, da hierdurch Blöcke und Pakete beschleunigt adressiert werden können.
- Der aktuell implementierte Architektur-Generator unterstützt lediglich die Basisfunktionalität zur Erzeugung ausführbarer Architekturmodelle. Er ist unter anderem um eine Layout- bzw. Autorouting-Funktion, Unterstützung hierarchischer Partitionen, Auflösung verlinkter Argumente, sowie eine Konfigurationsmöglichkeit zur Parametrisierung der Architekturkomponenten zu erweitern. Zudem muss eine Möglichkeit geschaffen werden vordefinierte Transitsysteme, wie beispielsweise Switches und Hubs, in Architekturmodelle zu integrieren. Hierbei reicht

die Definition eines Netzwerks durch Selektion von Partitionen, sowie der Festlegung von Protokollstapeln und eines Kanals nicht aus $(network = (p_1, ps_1), (p_2, ps_2), (p_3, ps_3), c_1)$. Vielmehr muss auch die Selektion und Konfiguration von Transitsystemen unterstützt werden $(network = (p_1, ps_1), (p_2, ps_2), (p_3, ps_3), new\_trans(p_4, ps_4, c_1))$, wobei diese wiederum Bestandteil einer bereits existierenden Partition sein können $(network = (p_1, ps_1), (p_2, ps_2), (p_3, ps_3), int\_trans(p_1, c_1))$.

- Alternativ zur Erweiterung des Architektur-Generators ist die Entwicklung eines universellen Generators für beliebigen Transformation von MLDesigner-Modellen denkbar, wodurch die Flexibilität erhöht wird. Hierzu ist eine steuernde Sprache inklusive Schnittstelle zu definieren, welche auch während der Modellausführung verwendbar sein muss, um so einen ausführbaren Entwurfsprozess zu ermöglichen.

- Die Idee des ausführbaren Entwurfsprozesses zur automatisierten Entwurfsraumanalyse sollte weiter konkretisiert werden. Zur Realisierung muss die den Entwurfsprozess abbildende Simulation Modelle erzeugen und in Form von Subsimulationen ausführen und bewerten können. Einen möglichen Ansatzpunkt zur Beschreibung und Steuerung eines solchen Prozesses stellen Simulation Sets dar. In diesem Zusammenhang muss auch der in Abschnitt 3.3 eingeführte Entwurfsprozess durch Ausführung validiert werden.

- Zur Auffindung eines optimalen Entwurfs können verschiedene, modellbasiert beschreibbare, Optimierungsverfahren verwendet werden. Um einen beschleunigten Austausch zu ermöglichen sollte eine standardisierte Optimierungsschnittstelle, sowie eine Bibliothek an Optimierungsverfahren entwickelt werden.

- In Bezug auf den Entwurf von Ausführungskomponenten sind weitere Experimente notwendig. Zum einen ist zu untersuchen, ob analog zu OSI bei den Kommunikationskomponenten auch hier ein Standard für den Entwurf eingeführt werden kann. Zum anderen muss geklärt werden, ob durch die Integration von Ausführungseinheiten die Scheduling-Reihenfolge bei der Modellausführung beeinflusst wird. Dies würde bedeuten, dass partitionierte Funktionen nur durch bestimmte Domänen beschrieben werden können.

- Momentan können die durch Funktionen zur plattformspezifischen Ausführung benötigten Ressourcen nicht automatisiert ermittelt werden. Aus diesem Grund sollten Bemessungsmechanismen, wie beispielsweise in [Lohfelder 2004] beschrieben, angebunden werden.

- Um ein Systemmodell zu validieren, muss die gewählte Realisierung verifiziert werden. Dies kann durch Ausführung des Systemmodells erreicht werden. Ein solcher Prozess könnte durch Einführung einer regelbasierten Verifikationssprache, wie in [Pacholik u. Fengler 2007] erläutert, beschleunigt werden.

- Die standardisierte Architekturbeschreibung dient als Grundlage für die plattformspezifische Synthese. Hierzu sollten weitere Experimente durchgeführt werden. Unter anderem ist zu klären, welche Domänen und Domänenkombinationen synthetisierbar sind. Auch sollte die aktuell auf C++ basierte Beschreibungsform von Primitives um eine alternative, synthetisierbare Sprache ergänzt werden.

Literaturverzeichnis

[Adamski 2001]
ADAMSKI, Dirk: Komponentenbasierte Simulation mechatronischer Systeme. In: *Fortschritts-Berichte VDI Reihe 10 Nr. 682* (2001)

[Alte 2005]
Altera Corp.: *Quartus 2 Project*. 2005. – http://www.altera.com/quartus2

[Baumann u. a. 2007a]
BAUMANN, Tommy ; HAUGUTH, Maik ; SALZWEDEL, Horst: Overcoming the Gap between Design at Electronic System Level (ESL) and Implementation for Networked Electronics. In: *Western MultiConference on Modeling & Simulation*, 2007

[Baumann u. a. 2008]
BAUMANN, Tommy ; HIPKINS, David ; MESTER, John: Model Based Design, Analysis and Validation of STEP using MLDesigner and SatLab / 50 Years Aeronautics and Astronautics, Stanford, USA. 2008. – Proceeding. http://aa.stanford.edu/aeroastro/50th/posters/model.pdf

[Baumann u. a. 2007b]
BAUMANN, Tommy ; PACHOLIK, Alexander ; SALZWEDEL, Horst: Performance Exploration with MLDesigner using Standardized Communication Interfaces. In: *DATE'07 Proceedings*, 2007

[Below 2003]
BELOW, Kai: *Methodology for Parameterisation of Large Scale Network Simulations*, Technische Universität Hamburg-Harburg, Diss., 2003. – http://www.tu-harburg.de/et6/papers/documents/Below_Kai/dissKaiBelow.pdf

[Bennett 1995]
BENNETT, B. S.: *Simulation Fundamentals*. Prentice Hall, 1995. – ISBN 0138132623

[Boehm 1988]
BOEHM, Barry W.: A Spiral Model of Software Development and Enhancement. In: *Computer* (1988). ISBN 0018–9162

[Buchenrieder 1999]
BUCHENRIEDER, Klaus: *Hardware/Software Codesign*. ITPress Verlag, 1999. – ISBN 3–929814–08–0

[Dally u. Towles 2004]
DALLY, William J. ; TOWLES, Brian: *Principles and Practices of Interconnection Networks*. Elsevier, 2004. – ISBN 0–12–200751–4

[Eck 2007]
ECK, Matthias: *Entwurf und Validierung des TCP/IP-Ethernet Protokolls auf Performance-Ebene*, Technische Universität Ilmenau, Diplomarbeit, 2007. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2007/Diplom2007_MatthiasEck.pdf

[Feldmann 1999]
FELDMANN, Martin: *Naturanaloge Verfahren*. Deutscher Universitätsverlag, 1999. – ISBN 3–8244–6890–5

[Fischer 2007]
FISCHER, Nils: *Entwurf einer Plug-and-Play Entwicklungsumgebung zur Optimierung vernetzter Avionik-Systemarchitekturen*, Technische Universität Ilmenau, Diplomarbeit, 2007. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2007/Diplom2007_NilsFischer.pdf

[Forsberg u. a. 1996]
FORSBERG, Kelvin ; MOOZ, Hal ; COTTERMAN, Howard: *Visualizing project management*. 1. John Wiley and Sons, Inc., 1996

[Givargis u. a. 2002]
GIVARGIS, Tony ; VAHID, Frank ; HENKEL, Jorg: System-Level Exploration for Pareto-Optimal Configurations in Parameterized Systems-on-a-Chip. In: *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2002, S. 416–422. – http://www.ics.uci.edu/~givargis/pubs/J8.pdf

[Glinz 2005]
GLINZ, M.: Rethinking the Notion of Non-Functional Requirements. In: *Proceedings of the Third World Congress for Software Quality* Bd. 2, 2005, S. 55–64

[Grimpe u. Oppenheimer 2003]
GRIMPE, Eike ; OPPENHEIMER, Frank: Extending the SystemC Synthesis Subset by Object Oriented Features. 2003. – Proceeding. http://www.offis.de/hs/publikationen

[Grötker u. a. 2003]
GRÖTKER, Thorsten ; LIAO, Stan ; MARTIN, Grant ; SWAN, Stuart: *System Design with SystemC*. Kluwer Academic Publishers Group, 2003. – ISBN 1–4020–7072–1

[Grüner 2006]
GRÜNER, David: *Kompositionsstrukturverhalten beim Systementwurf auf Missionsebene mit Hilfe der Unified Modelling Language und MLDesigner*, Technische Universität Ilmenau, Studienarbeit, 2006. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2006/Studienarbeit2006_Gruener.pdf

[Grüner 2007]
GRÜNER, David: *Entwurf eines Kernels für die Multidomain-Simulation*, Technische Universität Ilmenau, Diplomarbeit, 2007. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2007/Diplom2007_DavidGruener.pdf

[Hauguth 2000]
HAUGUTH, Maik: *Entwurf eines XML-basierten Dokumentenformates für Blockdiagramme*, Technische Universität Ilmenau, Diplomarbeit, 2000. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2000/Diplom2000_MaikHauguth.pdf

[IBM 2003]
IBM Inc.: *Rational Rose Realtime Project*. 2003. – http://www.ibm.com/software/rational

[Ingenieure Nov. 1996]
INGENIEURE, Verein D.: VDI Richtlinie 3633: Simulation von Logistik-, Materialfluß-, und Produktionssystemen, Begriffsdefinitionen. (Nov. 1996)

[Jackson 2001]
JACKSON, Michael: *Problem Frames: Analyzing and structuring software development problems.* Addison Wesley, 2001. – ISBN 9780201596274

[Jesser u. a. 2007]
JESSER, A. ; LÄMMERMANN, S. ; PACHOLIK, A. ; WEISS, R. ; RUF, J. ; FENGLER, W. ; HEDRICH, L. ; KROPF, T. ; ROSENSTIEL, W.: Analog Simulation Meets Digital Verification - A Formal Assertion Approach for Mixed-Signal Verification. In: *14. Workshop on Synthesis And System Integration of Mixed Information technologies (SASIM) Proceedings*, 2007

[Kalavade u. Lee 1996]
KALAVADE, Asawaree ; LEE, Edward A.: Complexity Management in System Level Design. In: *Journal of VLSI Signal Processing 14* Bd. 2, 1996, S. 157–169. – http://ptolemy.eecs.berkeley.edu/publications/papers/96/jvlsi-dmm/jvlsi_dmm.pdf

[Kurose u. Ross 2002]
KUROSE, J. F. ; ROSS, K. W.: *Computernetze – Ein Top-Down-Ansatz mit Schwerpunkt Internet.* Pearson Studium, 2002. – ISBN 3–8273–7017–5

[Lee 2002]
LEE, Edward A.: Embedded Software / Advances in Computers Vol. 56. 2002. – Proceeding. http://citeseer.ist.psu.edu/lee02embedded.html

[Lee u. Messerschmitt 1987]
LEE, Edward A. ; MESSERSCHMITT, David G.: Static scheduling of synchronous data flow programs for digital signal processing. In: *IEEE Trans. Comput. 36* 1 (1987). ISBN 0018–9340

[Liebezeit 2005]
LIEBEZEIT, Thomas V.: *Missionsbezogener modellgestützter Entwurf mobiler automatischer Systeme am Beispiel eines autonomen Unterwasserfahrzeugs*, Technische Universität Ilmenau, Diss., 2005. – http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2005/Dissertation2005_Liebezeit.pdf

[Lohfelder 2004]
LOHFELDER, Thomas: *Modeling Operating Systems in MLDesigner*, Technische Universität Ilmenau, Diplomarbeit, 2004. – http:

//wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2004/Diplom2004_ThomasLohfelder.pdf

[Lohse 2007]
LOHSE, Frank: *Entwurf und Validierung eines Planungssystems für das Freescale BeeKit mit dem Systementwurfswerkzeug MLDesigner*, Technische Universität Ilmenau, Diplomarbeit, 2007. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2007/Diplom2007_FrankLohse.pdf

[MLDe 2008]
MLDesign Technologies Inc.: *MLDesigner Project*. 2008. – http://www.mldesigner.com

[Moore 1965]
MOORE, G. E.: Cramming more components onto integrated circuits. In: *Electronics* 38 (1965), Nr. 8, S. 114–117

[NXP 2007]
NXP Semiconductors: *I2C-bus specification and user manual*. 2007. – http://www.nxp.com/acrobat_download/usermanuals/UM10204_3.pdf

[OMG 2004]
OMG: *UML 2.0 Superstructure Specification*. 2004. – http://www.omg.org/cgi-bin/doc?ptc/2004-10-02

[Open 2002]
OpenCores Organization: *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. 2002. – http://www.opencores.org/projects.cgi/web/wishbone/wishbone

[Pacholik u. Fengler 2007]
PACHOLIK, Alexander ; FENGLER, Wolfgang: A System Model for Formal Verification of TLM based Transaction Properties. In: *Proceedings of the tenth Communications and Networking Simulation Symposium (CNS)*, 2007. – ISBN 1–56555–312–8

[Paluch 2004]
PALUCH, Nils: *Anwendung von Co-Design für verteilte Echtzeitsysteme*, Technische Universität Ilmenau, Diplomarbeit, 2004. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2004/Diplom2004_NilsPaluch.pdf

[Phil 2002]
Philips Semiconductors: *SAA6588 RDS/RBDS pre-processor*. 2002. – http://www.nxp.com/acrobat_download/datasheets/SAA6588_2.pdf

[Pidd 1997]
Pidd, Michael: *Computer Simulation in Management Science*. Wiley & Sons, 1997. – ISBN 9780471979319

[Platzner u. Thiele 2006]
Platzner, Marco ; Thiele, Lothar: Vorlesungsskript Hardware/Software Codesign / ETH Zürich. 2006. – Proceeding

[Qi u. Moore 2002]
Qi, Honghui ; Moore, John B.: Direct Kalman filtering approach for GPS/INS integration. In: *IEEE Transactions on Aerospace and Electronic Systems*, 2002

[Rath u. a. 2006]
Rath, H. ; Schorcht, G. ; Salzwedel, H.: Simulationsumgebung für Bordnetze – Bordnetz-Spezifikationen modellieren und validieren. In: *Hanser automotive* (2006), May and June

[Rath u. Salzwedel 2004]
Rath, Holger ; Salzwedel, Horst: ANSI C Code Generation for ML-Designer Finite State Machines / 49. Internationales Wissenschaftliches Kolloquium IWK'2004, Ilmenau. 2004. – Proceeding

[Rath 2007]
Rath, Michael: *Generierung von Architekturmodellen aus Verhaltensmodellen basierend auf dem Network Block Set im MLDesigner*, Technische Universität Ilmenau, Studienarbeit, 2007

[Rech 2008]
Rech, Jörg: *Ethernet – Technologien und Protokolle für die Computervernetzung*. 2. Heise, 2008. – ISBN 978–3–936931–40–2

[Royce 1987]
Royce, W. W.: Managing the development of large software systems: concepts and techniques. In: *ICSE 1987: Proceedings of the 9th international conference on Software Engineering* (1987), S. 328–338. ISBN 0–89791–216–0

[Salzwedel 2004a]
SALZWEDEL, H.: Mission Level Design of Avionics. In: *AIAA-IEEE DASC 04, 23rd Digital Avionics Systems Conference*, 2004

[Salzwedel 2004b]
SALZWEDEL, Horst: Design Technology Development Towards Mission Level Design / 49. Internationales Wissenschaftliches Kolloquium IWK'2004, Ilmenau. 2004. – Proceeding

[Salzwedel 2007]
SALZWEDEL, Horst: Complex System Design Automation In The Presence Of Bounded And Statistical Uncertainties / 52. Internationales Wissenschaftliches Kolloquium IWK'2007, Ilmenau. 2007. – Proceeding

[Salzwedel u. a. 2008]
SALZWEDEL, Horst ; FISCHER, Nils ; BAUMANN, Tommy: Aircraft Level Optimization of Avionics Architectures. In: *AIAA-IEEE DASC 08, 27th Digital Avionics Systems Conference*, 2008

[Salzwedel u. a. 2009]
SALZWEDEL, Horst ; FISCHER, Nils ; SCHORCHT, Gunar: Moving Design Automation of Networked Systems to Early Vehicle Level Design Stages. In: *SAE 2009 World Congress*, 2009

[Sangiovanni-Vincentelli u. Martin 2001]
SANGIOVANNI-VINCENTELLI, Alberto ; MARTIN, Grant: A Vision for Embedded Software / University of California at Berkeley. 2001. – Proceeding

[Schienmann 2002]
SCHIENMANN, Bruno: *Kontinuierliches Anforderungsmanagement.* Addison-Wesley, 2002. – ISBN 3–8273–17–87–8

[Schneider u. a. 2005]
SCHNEIDER, Herfried M. ; BUZACOTT, John A. ; RÜCKER, Thomas: *Operative Produktionsplanung und -steuerung.* Oldenbourg Wissenschaftsverlag, 2005. – ISBN 978–3–486–57691–7

[Schorcht 2000]
SCHORCHT, Gunar: *Entwurf integrierter Mobilkommunikationssysteme auf Missionsebene.* Logos Verlag, 2000. – ISBN 3–89722–462–3

[Sikora u. Drechsler 2002]
SIKORA, Axel ; DRECHSLER, Rolf: *Software-Engineering und Hardware-Design Eine systematische Einführung*. Carl Hanser Verlag, 2002. – ISBN 3–446–21861–0

[Soltero u. a. 2002]
SOLTERO, Manny ; ZHANG, Jing ; COCKRILL, Chris: 422 and 485 Standards Overview and System Configurations / Texas Instruments Incorporated. 2002. – Proceeding. http://focus.ti.com/lit/an/slla070c/slla070c.pdf

[Spaniol u. Jakobs 1993]
SPANIOL, Otto ; JAKOBS, Kai: *Rechnerkommunikation – OSI-Referenzmodell, Dienste und Protokolle*. VDI-Verlag, 1993. – ISBN 3–18–401207–7

[Tannenbaum 2003]
TANNENBAUM, A. S.: *Computernetzwerke*. Pearson Studium, 2003. – ISBN 978–3827370464

[Trim 1999]
Trimble Navigation Limited: *TSIP Reference*. 1999. – http://trl.trimble.com/docushare/dsweb/Get/Document-6222/TSIP%20Reference%20Manual%20-%20Rev%20C.pdf

[Univ 1998]
University of California at Berkeley: *Ptolemy Classic Project*. 1998. – http://ptolemy.eecs.berkeley.edu/ptolemyclassic

[Univ 2005]
University of California at Berkeley: *Ptolemy II Project*. 2005. – http://ptolemy.eecs.berkeley.edu/ptolemyII

[Vanderperren u. Dehaene 2005]
VANDERPERREN, Yves ; DEHAENE, Wim: UML 2 and SysML: An Approach to Deal with Complexity in SoC/NoC Design. In: *DATE'05 Proceedings*, 2005. – ISBN 0–7695–2288–2

[Wideman 2003]
WIDEMAN, R. M.: Software Development and Linearity. In: *Projects & Profits Special Issue* (2003), March, S. 18

[Zeigler u. a. 2000]
ZEIGLER, Bernard P. ; KIM, Tag G. ; PRAEHOFER, Herbert: *Theory*

of Modeling and Simulation. Orlando, FL, USA, Academic Press, Inc., 2000. – ISBN 0127784551

[Zens 2003]
ZENS, Matthias: *Entwicklung von Methoden für die Analyse von elektronischen Ventilsteuerungssystemarchitekturen*, Technische Universität Ilmenau, Diplomarbeit, 2003. – http://wcms1.rz.tu-ilmenau.de/fakia/fileadmin/template/startIA/sst/Diplomarbeiten/2003/Diplom2003_MatthiasZens.pdf

[Zentner 2006]
ZENTNER, John M.: *A Design Space Exploration Process for Large Scale, Multi-objective Computer Simulations*, Georgia Institute of Technology, Diss., 2006. – http://etd.gatech.edu/theses/submitted/etd-07072006-105333/unrestricted/zentner_john_marc_200608_phd.pdf

[Zimmermann u. Schmidgall 2007]
ZIMMERMANN, Werner ; SCHMIDGALL, Ralf: *Bussysteme in der Fahrzeugtechnik – Protokolle und Standards*. Vieweg, 2007. – ISBN 978–3–8348–0235–4

Anhang A

Hauptschnittstelle des ESL-Targets

```
#ifndef ESL_ESLTARGET_HPP
#define ESL_ESLTARGET_HPP

#include <list>
#include <map>

#include "DETarget.h"
#include "StringState.h"

class Channel;
class ProtocolStack;
class Partition;
class Interference;
class DEPortHole;
class InDEPort;
class OutDEPort;
class QString;
class Logging;
class Monitor;

class ESLTarget: public DETarget
{
  public:
    ESLTarget();
    ~ESLTarget();

    Block* makeNew() const;
    bool getDebugState();
    bool getMonitorState();
```

```
  Channel* getChannel(Block* pBlock);
  Channel* getAssoziatedChannel(Block* pBlock);
  ProtocolStack* getProtocolStack(Block* pBlock);
  std::list<Channel*>& getChannelList() { return
     mChannelList; }
  std::list<ProtocolStack*>& getProtocolStackList() {
     return mProtocolStackList; }
  std::list<Partition*>& getPartitionList() { return
     mPartitionList; }
  DEPortHole* getPortByFullName(Kernel::String
     pPortFullName);
  std::list<InDEPort*> getMappedReceiverPortList(OutDEPort*
      pSenderPort);
  float getInterferenceFactor(Channel* pChannel);
  float getLoggingInterval();
  Logging* getLogging(int& pIdentifier);
  std::list<Logging*>& getLoggingList() { return
     mLoggingList; }

  int addValueLogging(Block* pBlock, Kernel::String
     pEntryName);
  int addAverageLogging(Block* pBlock, Kernel::String
     pEntryName, float pMinimum = 0, float pMaximum = 1);
  int addMatrixLogging(Kernel::String pGroupName, Kernel::
     String pParticipantName);
  void updateValueLogging(int& pIdentifier, Kernel::String
     pIntervalValue);
  void updateAverageLogging(int& pIdentifier, float
     pIntervalActivity);
  void updateMatrixLogging(int& pIdentifier, Kernel::String
     & pReceiver, std::map<Kernel::String, int>
     pSenderIntervalActivityMap);

private:
  StringState mESLSetupStringState;
  EnumState mESLDebugEnumState;
  EnumState mESLMonitorEnumState;
  std::list<Channel*> mChannelList;
  std::list<ProtocolStack*> mProtocolStackList;
  std::list<Partition*> mPartitionList;
  std::multimap<OutDEPort*, InDEPort*>
     mSenderReceiverMultiMap;
  std::list<Interference> mInterferenceList;
  float mLoggingInterval;
  std::list<Logging*> mLoggingList;
  Monitor* mMonitor;

  typedef enum BooleanT
  {
```

```
      DISABLED = 0,
      ENABLED  = 1
    };

    void setup();
    void cleanup();
    void initializeArchitectureModel(Galaxy& pGalaxy);
    void resetLogging();
    static void verificationCallback(Star* pStar, const char*
        pTextArgument);
};

#endif
```

Anhang B

Auszug aus dem Architecture Block Set

ArchitectureBlockSet library

Date: Sat May 3 11:01:47 2008
Version: 0.0 01/18/2007
Author: ketzman

- **Current Location:**
- MLD_USER
- ArchitectureBlockSet

Models:

- Base
- BeeKitProtocols
- CAN
- EIA232
- GeneratorExample
- TCPIPEthernet
- Test_CPU
- Test_Ethernet
- Wishbone

Composite Data Structures:

Root.ArchitectureBlockSet			
Root.ArchitectureBlockSet.ChannelPacket			
Data	Root		
Delay	Root.Float	[0, Inf)	0
SenderBlock	Root.Pointer		
DispatchMode	Root.ENUM.DispatchMode		SendDelayed
Root.ArchitectureBlockSet.ExecuterPacket			
Data	Root.Pointer		
SenderPort	Root.Pointer		
ReceiverPort	Root.Pointer		
Root.ArchitectureBlockSet.FunctionPacket			
Data	Root.Pointer		
Size	Root.Integer	(0, Inf)	1
ArrivalTime	Root.Float	[0, Inf)	0
ReceiverPort	Root.Pointer		

Enumeration Data Structures:

Root.ENUM.DispatchMode	
0	SendDelayed
1	SendUndelayed
2	DelayOnly

Generated on Mon Jun 23 2008 08:21:36 by **MLDesigner** v2.7.r00 - *The Next Generation System Design Tool*

Base library

Date: Sat May 3 11:01:47 2008
Version: 0.0 01/30/2007
Author: ketzman

- **Current Location:**
- MLD_USER
- ArchitectureBlockSet
- Base

Import Models:

ArchitectureBlockSet

Models:

AverageLogging	This primitive announces a average logging with name of parameter EntryName to the ESL target and updates it with the data receiving on port Data. The interval for average computation is defined by the parameters Minimum and Maximum.

ChannelInterface	This primitive provides a standardized interface for the design of communication channels, whereby each channel represents the physical connection (OSI Physical Layer or abstraction) between partition protocols. The received packets are sent to all connected protocol blocks inclusive or exclusive the sender protocol. This can be justified via parameter SenderEcho. The count of connected partition interfaces is tested to be in range of [MinParticipants, MaxParticipants]. The value -1 disables checking.
ExecuterInterface	
ProtocolChannelInterface	This primitive provides a standardized protocol stack interface in channel direction. Each protocol stack implements all OSI layers without Application Layer and Physical Layer. It maps sender to receiver ports converts the functional data flow to an architectural and vice versa.
ProtocolFunctionInterface	This primitive provides a standardized protocol stack interface in functional direction. Each protocol stack implements all OSI layers without Application Layer and Physical Layer. Task is to transform received data particles to FunctionPackets and vice versa. If mapping is enabled (parameter Mapping), FunctionPackets with receiver port informations are created and sent exactly to these ports. Therefore mapping entries have to be defined via the ESLTarget. If mapping is disabled, alle received data particles are send to all connected ports of the receiving ProtocolFunctionInterface. With parameter Delay a delay for all FunctionPackets to sent can be determined. Parameter FixedDataSize determines a fixed value for the size member of those packets. If the value is lower or equal than zero the original data size is computed and used.
ValueLogging	This primitive announces a value logging with name of parameter EntryName to the ESL target and updates it with the data receiving on port Data.

Generated on Mon Jun 23 2008 07:42:03 by **MLDesigner** v2.7.r00 - *The Next Generation System Design Tool*

TCPIPEthernet library

Date: Sat May 3 11:02:58 2008
Version: 0.0 03/05/2007
Author: mec

- **Current Location:**
- MLD_USER
- ArchitectureBlockSet
- TCPIPEthernet

Models:

EthernetChannel	This star is a module, that represents the medium used for network transfer. The data will be delayed according to the parameters given. ChannelInterface is a standardized component from the ArchitectureBlockSet.
Examples	

Layers
Sockets
TCPIPProtocol

Composite Data Structures:

Root.TcpIpEthernet

Root.TcpIpEthernet.ARPPacket

SenderMAC	Root.String		0
ReceiverMAC	Root.String		0
SenderIP	Root.String		0
ReceiverIP	Root.String		0
OperationCode	Root.Integer	(0, Inf)	1
Size	Root.Integer	[0, Inf)	0

Root.TcpIpEthernet.DataPacket

Data	Root		
Size	Root.Integer	(0, Inf)	1
ArrivalTime	Root.Float	[0, Inf)	0

Root.TcpIpEthernet.EthernetPacket

Data	Root		
Payload	Root.Integer	(0, Inf)	0
SenderMAC	Root.String		0
ReceiverMAC	Root.String		0
Overhead	Root.Integer	(0, Inf)	1

Root.TcpIpEthernet.IPPacket

Data	Root		
Overhead	Root.Integer	[0, Inf)	0
ReceiverIP	Root.String		0
SenderIP	Root.String		0

Root.TcpIpEthernet.Jam

SenderMAC	Root.String		0
ReceiverMAC	Root.String		FFFFFFFFFFFF
Data	Root.String		AAAAAAAAAAAA
Overhead	Root.Integer	[0, Inf)	0
Payload	Root.Integer	[0, Inf)	6

Root.TcpIpEthernet.SAPPacket

ReqIP	Root.String		0
ReqMAC	Root.String		0
isMy	Root.Integer	[0, 1]	0

Root.TcpIpEthernet.TCPPacket

Packet	Root.Integer	(0, Inf)	1
seqNo	Root.Integer		0
ECN	Root.Integer		0
ECN_ON	Root.Integer		0

ackNo	Root.Integer		0
winNo	Root.Integer		0
sizeNo	Root.Integer		0
numNo	Root.Integer		0
LAckNo	Root.Integer		0
Time	Root.Float		0
size	Root.Integer		0
source	Root.Integer		0
destination	Root.Integer		0
VC	Root.Integer		0
Data	Root		
isFin	Root.Integer	[0, 1]	0
Overhead	Root.Integer	[0, Inf)	0
Payload	Root.Integer	[0, Inf)	0
isAck	Root.Integer	[0, 1]	0
isSyn	Root.Integer	[0, 1]	0
SenderPort	Root.Integer	[0, Inf)	0
ReceiverPort	Root.Integer	[0, Inf)	0
Root.TcpIpEthernet.TCPPacket_			
SenderPort	Root.Integer		0
ReceiverPort	Root.Integer		0
Overhead	Root.Integer		0
Data	Root		
seqNo	Root.Integer		0
ackNo	Root.Integer		0
winSize	Root.Integer		0
Payload	Root.Integer		0
Size	Root.Integer		0
isFin	Root.Integer	[0, 1]	0
isAck	Root.Integer	[0, 1]	0
Time	Root.Float		0
numNo	Root.Integer		0

Generated on Mon Jun 23 2008 08:07:14 by **MLDesigner** v2.7.r00 - *The Next Generation System Design Tool*

Anhang C

Konfigurations-Template des Architektur-Generators

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<convertconfig version="1.0" xmlns:xsi="http://www.w3.org
   /2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation=""
   >

  <!--
   variable definitions
   * variables MLD and and MLD_USER are necessary
-->

  <variable name="MLD" value="/usr/local/mld2"/>

  <variable name="MLD_USER" value="/home/username/MLD"/>

  <!--
   input files
   * interface     : path to channel interface model
   * model         : path to main system
   * defaultchannel: path to default channel (optional)
-->

  <input type="defaultchannel" url="file:$MLD_USER/path/to/
     model/model.mml"/>

  <input type="interface" url="file:$MLD_USER/path/to/model/
     model.mml"/>

  <input type="model" url="file:$MLD_USER/path/to/model/model
     .mml"/>
```

```
  <!--
   output library
   * the library is placed directly under directory $MLD_USER
   * overwrite:
     * 'true', '1' : if destination directory exists and is
        not empty, the content
                     is erased and rewritten by the converter
     * 'false','0' : if destination directory exists and is
        not empty, the converter
                     will abort
-->

  <outputlibrary name="arch_model" overwrite="false"/>

  <!--
   partition mapping
   * this section is optional and can be ommitted (
      annotations are used in this case)
   * mode:
     * 'annotation': parse model annotation propertys
     * 'this'      : use these partition mappings
   * sub tags are named 'map'
     * attribute 'element'  : object path within model
     * attribute 'partition': the partition this object
        should be mapped to
-->

  <partitionmapping mode="annotation">
    <map element="system.instance#0" partition="A"/>
    <map element="system.instance#1" partition="A"/>
  </partitionmapping>

  <!--
   network definitions
   * this section is optional and can be ommitted (
      annotations are used in this case)
   * mode:
     * 'annotation': parse model annotation property
     * 'this'      : use these networks
   * sub tags are named 'network'
     * attribute 'partitions': comma separated list of
        partition names
     * attribute 'channelurl': url to channel model

   NOTE: Networks that occure in the funtional model which
      are not listed here
   (or inside the system annotation) are sythesized during
      runtime using the
   default channel (see above).
```

```
-->

  <networks mode="annotation">
    <network channelurl="file:$MLD_USER/path/to/channel/model
       /model.mml" partitions="A, B, C"/>
  </networks>

</convertconfig>
```

Anhang D

VHDL-Beschreibung des Protocol Encoders

```
-- IP Core ProtocolEncoder
--
-- Conforms to Wishbone SoC Architecture Specification,
   Revision B.3,
-- Released September 7, 2002
--
-- Date:    Tuesday 16 May 2006 16:40:26+02:00
-- Version: 0.0 11/30/2005
-- Author:  hauguth

-- The IP core acts as a Wishbone slave interface.

library ieee;
use ieee.std_logic_1164.all;


entity ProtocolEncoder is

  -- Generic parameters
  generic (
    TestParameter : integer := 88; -- only for Testing
    testparam2 : real := 0.0 --
  );

  port (
    -- WISHBONE ports
    WB_CLK_I, WB_RST_I: in std_logic;
    WB_ADR_I: in std_logic_vector (2 downto 0); -- Used for
       register selection
    WB_WE_I:  in std_logic; -- Write or read cycle selection
```

```
    WB_DAT_I: in std_logic_vector (7 downto 0); -- Data input
    WB_DAT_O: out std_logic_vector (7 downto 0); -- Data
       output
    WB_ACK_O: out std_logic; -- Acknowledge of a transfer
    WB_CYC_I: in  std_logic; -- A bus cycle is in progress
    WB_STB_I: in  std_logic; -- Specifies transfer cycle (
       single read/write, block transfer, RMW)

    -- Interrupt request register for signaling events
    IRQ_O: out std_logic; -- causes master to interrupt

    -- External (off-chip) connections
    PaketData: out   std_logic_vector( 7 downto 0);
    PaketData_stb: out std_logic;
    PaketData_ack: in std_logic; --
    TXRdy: in    std_logic -- Transmission ready signal from
       UART

    -- Internal non WISHBONE signals
  );

  -- Other entity declarations
end ProtocolEncoder;

architecture ProtocolEncoder1 of ProtocolEncoder is
  -- Wishbone Registers
  signal   Start_reg: std_logic_vector( 7 downto 0 );
  signal   End_reg: std_logic;
  signal   Data_reg: std_logic_vector( 7 downto 0 );
  signal   Rdy_reg: std_logic;
  signal   Status_reg: std_logic;

  -- write event indicators
  signal Start_reg_write : std_logic;
  signal End_reg_write : std_logic;
  signal Data_reg_write : std_logic;

  -- Wishbone adress constants
  constant Start_adr: std_logic_vector(2 downto 0) := B"000";
  constant End_adr: std_logic_vector(2 downto 0) := B"001";
  constant Data_adr: std_logic_vector(2 downto 0) := B"010";
  constant Rdy_adr: std_logic_vector(2 downto 0) := B"011";
  constant Status_adr: std_logic_vector(2 downto 0) := B"100"
     ;

  -- Constants for status register flags
  constant Rdy_reg_IRQ : std_logic_vector(0 downto 0 ) := "
     1000";
```

```
  -- Internal signals
  signal busy : std_logic; -- set if no I/O must be made


  -- Uncomment xyz_2 state if transitions depend on response
     signals
  -- of the output of xyz state
  type StateT is (
    Init,  -- Init_2,
    Data,  Data_2,
    StuffByte,  -- StuffByte_2,
    Ende_DLE,  -- Ende_DLE_2,
    Ende_ETX,  -- Ende_ETX_2,
    Start_DLE,  -- Start_DLE_2,
    Sent,  -- Sent_2,
    Sent1,  -- Sent1_2,
    Sent2 -- Sent2_2
  );

  signal CurrentState: StateT;

begin

  -- simple asynchrounous (!) handshaking protocol without
     waitstates.
  -- see WISHBONE Spec. Permission 3.10
  -- for a synchronous implementation, wait states or
     registered
  -- feedback bus cycles, a more sophisticated scheme
  -- must be used instead
  WB_ACK_O <= WB_STB_I and WB_CYC_I and not busy;

  -- simple synchronous handshaking protocol without
     waitstates.
  -- see WISHBONE Spec. for more details
  -- assign_WB_ACK_O: process( WB_CLK_I )
  -- begin
  --   if rising_edge(WB_CLK_I) then
  --     WB_ACK_O <= WB_CYC_I and WB_STB_I not busy;
  --   end if;
  -- end process assign_WB_ACK_O;

  -- assign WB_DAT_O;
  assign_WB_DAT_O: process( WB_CLK_I )
  begin
    if falling_edge(WB_CLK_I) then
      case WB_ADR_I is
        when Rdy_adr =>
          WB_DAT_O(0) <= Rdy_reg;
```

```
          WB_DAT_O(7 downto 1) <= (others => '0');
        when Status_adr =>
          WB_DAT_O(0) <= Status_reg;
          WB_DAT_O(7 downto 1) <= (others => '0');
        when others => WB_DAT_O <= (others => '0');
      end case;
    end if;
  end process assign_WB_DAT_O;

  -- reset and WISHBONE write access
  ResetWriteAccess: process( WB_CLK_I, WB_RST_I)
  begin
    if WB_CLK_I'event and WB_CLK_I = '1' then
      if WB_RST_I = '1' then
         busy <= '0';
      elsif (WB_CYC_I = '1') and (WB_STB_I = '1') and (
         WB_WE_I = '1') then
          Start_reg_write <= '0';
          End_reg_write <= '0';
          Data_reg_write <= '0';
        case WB_ADR_I is
          when Start_adr =>
            Start_reg(7 downto 0) <= WB_DAT_I(7 downto 0);
            Start_reg_write <= '1';
          when End_adr =>
            End_reg <= WB_DAT_I(0);
            End_reg_write <= '1';
          when Data_adr =>
            Data_reg(7 downto 0) <= WB_DAT_I(7 downto 0);
            Data_reg_write <= '1';
          when others => null;
        end case;
      end if;
    end if; -- clock edge
  end process ResetWriteAccess;

ProtocolEncoder_FSM: process(WB_RST_I, WB_CLK_I)
  begin
    if WB_RST_I ='1' then
      -- initialization on Reset condition
      CurrentState <= Init;
    elsif rising_edge(WB_CLK_I) then
      CASE CurrentState IS
        WHEN Init =>
          -- CurrentState <= Init_2;  -- uncomment if
             transitions depend on response to output
          -- WHEN Init_2 =>
          IF Start_reg_write = '1' THEN
            CurrentState <= Start_DLE;
```

```
        END IF;
      WHEN Data =>
        -- TODO: implement entry action WriteOutput(
           PaketData, ReadNewInput(Data));
        PaketData_reg <= Data_reg;
        -- CurrentState <= Data_2;  -- uncomment if
           transitions depend on response to output
      WHEN Data_2 =>
        IF TXRdy THEN
          CurrentState <= Sent1;
        END IF;
      WHEN StuffByte =>
        -- TODO: implement entry action WriteOutput(
           PaketData, 0x10);
        -- CurrentState <= StuffByte_2;  -- uncomment if
           transitions depend on response to output
        -- WHEN StuffByte_2 =>
        IF TXRdy THEN
          CurrentState <= Sent2;
        END IF;
      WHEN Ende_DLE =>
        -- TODO: implement entry action WriteOutput(
           PaketData, 0x10);
        -- CurrentState <= Ende_DLE_2;  -- uncomment if
           transitions depend on response to output
        -- WHEN Ende_DLE_2 =>
        IF TXRdy THEN
          CurrentState <= Ende_ETX;
        END IF;
      WHEN Ende_ETX =>
        -- TODO: implement entry action WriteOutput(
           PaketData, 0x03);
        -- CurrentState <= Ende_ETX_2;  -- uncomment if
           transitions depend on response to output
        -- WHEN Ende_ETX_2 =>
        IF TXRdy THEN
          CurrentState <= Init;
        END IF;
      WHEN Start_DLE =>
        -- TODO: implement entry action WriteOutput(
           PaketData, 0x10);
        -- CurrentState <= Start_DLE_2;  -- uncomment if
           transitions depend on response to output
        -- WHEN Start_DLE_2 =>
        IF TXRdy THEN
          CurrentState <= Sent;
        END IF;
      WHEN Sent =>
        -- TODO: implement entry action WriteOutput(Rdy, 1)
```

```
            ;
          -- CurrentState <= Sent_2;  -- uncomment if
             transitions depend on response to output
          -- WHEN Sent_2 =>
          IF Data_reg_write = '1' THEN
            CurrentState <= Data;
          END IF;
        WHEN Sent1 =>
          -- TODO: implement entry action WriteOutput(Rdy, 1)
             ;
          -- CurrentState <= Sent1_2;  -- uncomment if
             transitions depend on response to output
          -- WHEN Sent1_2 =>
          IF Data_reg_write = '1'
          -- TODO: implement guard condition int(ReadNewInput
             (Data)) != 0x10
          THEN
            CurrentState <= Data;
          ELSIF Data_reg_write = '1'
          -- TODO: implement guard condition int(ReadNewInput
             (Data)) == 0x10
          THEN
            CurrentState <= StuffByte;
          ELSIF End_reg_write = '1' THEN
            CurrentState <= Ende_DLE;
          END IF;
        WHEN Sent2 =>
          -- CurrentState <= Sent2_2;  -- uncomment if
             transitions depend on response to output
          -- WHEN Sent2_2 =>
            CurrentState <= Data;
        WHEN OTHERS => CurrentState <= Init;
      END CASE;
    end if;
  end process ProtocolEncoder_FSM;

end ProtocolEncoder1;
```