Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Universität Dortmund
am Fachbereich Informatik

von

Peter Herrmann

Dortmund
1997

Kurzfassung

Der korrektheitssichernde Entwurf von Kommunikationsprotokollen für den Hochleistungsdatentransfer wird durch das sogenannte Transferprotokoll-Framework unterstützt. Es besteht aus einer Sammlung von Spezifikationsbausteinen und Theoremen. Die Spezifikationsbausteine beschreiben einzelne Protokollmechanismen sowie Eigenschaften von Kommunikationsdiensten. Man modelliert Protokolle unter vergleichsweise geringem Aufwand, da die Protokollmechanismen auf einfache Art durch Spezifikationsbausteine dargestellt werden können. Diese Teilspezifikationen werden miteinander zu der Protokollspezifikation komponiert. Unter Verwendung der Spezifikationsbausteine für die Diensteigenschaften erstellt man auf die gleiche Weise Dienstspezifikationen.

Der Nachweis, daß ein spezifiziertes Kommunikationsprotokoll einen Kommunikationsdienst erbringt, kann aufgrund des modularen Aufbaus der Spezifikationen in Einzelbeweise zerlegt werden. In jedem Einzelbeweis wird nachgewiesen, daß ein Subsystem des Protokolls eine einzelne Diensteigenschaft erfüllt. Die Theoreme des Frameworks entsprechen diesen Einzelbeweisen.

Die Spezifikationsbausteine des Frameworks sind in cTLA$^+$ formuliert, das auf Leslie Lamports Temporal Logic of Actions (TLA) basiert und sich speziell für die Spezifikation von Prozeßsystemen eignet.

Inhaltsverzeichnis

1 Einleitung **1**

2 Hochleistungsprotokolle **7**
2.1 Grundlagen . . . 8
2.2 Transportprotokolle . . . 10
2.2.1 Der Transportdienst . . . 11
2.2.2 Protokollfunktionen des Datentransfers . . . 11
2.2.3 Protokollfunktionen der Verbindungsverwaltung . . . 13
2.2.4 Transportprotokoll-Architekturen . . . 14
2.3 Hochleistungsprotokollarchitekturen . . . 15
2.3.1 Anwendungen von Hochleistungsnetzen . . . 16
2.3.2 Optimierung konventioneller Protokolle . . . 16
2.3.3 Adaption der Protokollfunktionen . . . 17
2.3.4 Optimierung der Protokollgraphen . . . 19
2.3.5 Protokollkonfigurierung . . . 21
2.3.6 Parallelisierung von Protokollfunktionen . . . 22
2.4 Einige Beispiele von Hochleistungsprotokollen . . . 22
2.4.1 Übertragungsverfahren . . . 22
2.4.2 NETBLT . . . 24
2.4.3 VMTP . . . 24
2.4.4 XTP . . . 25
2.4.5 SNR . . . 26
2.4.6 PATROCLOS . . . 26
2.4.7 MSP . . . 26
2.5 Einige Gedanken zur Modellierung von Hochleistungsprotokollen . . . 27

3 Formale und funktionale Entwurfsmethoden **29**
3.1 Grundlagen . . . 29
3.1.1 Anwendungsgebiet . . . 30
3.1.2 Modellbildung . . . 31
3.1.3 Notation . . . 31
3.1.4 Systembildung . . . 33
3.1.5 Verifikationsmethoden . . . 34
3.2 Formale Protokollentwicklung . . . 36
3.2.1 Formale Beschreibungs- und Analysetechniken . . . 36

3.2.2 Anwendung formaler Methoden auf Hochleistungstransportprotokolle 38
3.2.3 Anforderungen an eine Spezifikations- und Verifikationstechnik für Hochleistungsprotokolle . . . 39
3.3 TLA . . . 40
3.3.1 Syntax und Semantik . . . 40
3.3.2 Spezifikation . . . 46
3.3.3 Verifikation . . . 48
3.3.4 Refinement Mappings . . . 53
3.3.5 Komposition . . . 55
3.4 TLA$^+$ und tTLA$^+$ . . . 57
3.4.1 TLA$^+$ . . . 57
3.4.2 tTLA$^+$ . . . 60
3.5 Werkzeugunterstützung . . . 62
3.5.1 Unterstützung der Spezifikationsentwicklung . . . 62
3.5.2 Unterstützung der Verifikation . . . 63

4 Kompositionale Spezifikation und strukturierte Verifikation 65
4.1 Der kompositionale Spezifikationsstil . . . 66
4.1.1 Prozeßmodell . . . 67
4.1.2 Prozeßkopplung . . . 68
4.1.3 Stilkonventionen . . . 71
4.1.4 Nachweis der Erfüllbarkeit komponierter Systeme . . . 72
4.2 Strukturierte Verifikation . . . 76
4.2.1 Subsysteme . . . 77
4.2.2 Nachweis von Kopplungsbeschreibungen . . . 78
4.3 Spezifikationen in cTLA$^+$ . . . 79
4.3.1 cTLA$^+$-Prozeßtypspezifikationen . . . 79
4.3.2 cTLA$^+$-Prozeßoperationen . . . 81
4.3.3 Einhaltung der Stilkonventionen . . . 91
4.3.4 Strukturierung von Fairnessbeweisen . . . 92

5 Ein Framework für Hochleistungstransportprotokolle 95
5.1 Aufbau des Frameworks . . . 96
5.1.1 Die Bibliotheken des Frameworks . . . 96
5.1.2 Die Struktur der Frameworkbibliotheken . . . 98
5.1.3 Der Aufbau der Spezifikationsbausteine . . . 100
5.1.4 Die Syntax der Theoreme . . . 102
5.2 Service Constraints (SCs) . . . 103
5.2.1 Safety-SCs des Datentransfers . . . 103
5.2.2 Liveness-SCs des Datentransfers . . . 106
5.2.3 Transfer von Signalisierungsdaten . . . 110
5.2.4 Safety-SCs der Intraconnection-Coordination . . . 111
5.2.5 Liveness-SCs der Intraconnection-Coordination . . . 114
5.2.6 SCs der Interconnection-Coordination . . . 115
5.3 Abstract Protocol Mechanisms (APMs) . . . 117
5.3.1 Safety-APMs des Datentransfers . . . 118

5.3.2 Liveness-APMs des Datentransfers 124
5.3.3 APMs der Intraconnection-Coordination 128
5.3.4 APMs der Interconnection-Coordination 129
5.4 Abstract Media Constraints (AMCs) 129
5.5 Finite Abstract Protocol Mechanisms (FAPMs) 130
5.5.1 Entwurf der FAPMs des Datentransfers und der Intraconnection-Coordination 130
5.5.2 Entwurf der FAPMs der Interconnection-Coordination 138
5.6 Theoreme zu den SCs 141
5.6.1 Theoreme zu den Safety-SCs des Datentransfers 141
5.6.2 Theoreme zu den Liveness-SCs des Datentransfers 145
5.6.3 Theoreme zu den SCs der Intraconnection-Coordination 151
5.7 Theoreme zu den APMs 152
5.7.1 Theoreme zu den APMs des Datentransfers und der Intraconnection-Coordination 152
5.7.2 Theoreme zu den APMs der Interconnection-Coordination 154

6 Eine Anwendung: XTP 4.0 **155**
6.1 Übersicht 156
6.1.1 Protokollfunktionen des Datentransfers 156
6.1.2 Protokollfunktionen der Verbindungsverwaltung 157
6.1.3 XTP-Protokolldateneinheiten 160
6.2 Spezifikation des XTP-Protokolls 162
6.2.1 Aufbau der Protokollspezifikation 164
6.2.2 Instantiierung der FAPMs des Datentransfers 165
6.2.3 Komposition der FAPMs des Datentransfers 172
6.2.4 Instantiierung der FAPMs der Signalisierungskanäle 173
6.2.5 Komposition der FAPMs der Signalisierungskanäle 180
6.2.6 Instantiierung der FAPMs der Intraconnection-Coordination 181
6.2.7 Komposition der FAPMs der Intraconnection-Coordination 184
6.2.8 Instantiierung der FAPMs der Interconnection-Coordination 185
6.2.9 Komposition der FAPMs der Interconnection-Coordination 186
6.2.10 Instantiierung der AMCs 186
6.2.11 Instantiierung und Komposition der APMs 188
6.3 Spezifikation des Transportdienstes 188
6.3.1 Instantiierung der SCs des Datentransfers 188
6.3.2 Komposition der SCs des Datentransfers 189
6.3.3 Instantiierung der SCs der Signalisierungskanäle 190
6.3.4 Komposition der SCs der Signalisierungskanäle 194
6.3.5 Instantiierung der SCs der Intraconnection-Coordination 194
6.3.6 Komposition der SCs der Intraconnection-Coordination 194
6.3.7 Instantiierung der SCs der Interconnection-Coordination 195
6.3.8 Komposition der SCs der Interconnection-Coordination 195
6.4 Verifikation der Dienstspezifikation 195
6.4.1 Verifikation der APMs 195
6.4.2 Verifikation der SCs 197

6.5 Bewertung der XTP-Qualität 198
6.5.1 Eigenschaften der SCs 199
6.5.2 Konformität der FAPMs 200
6.5.3 Unterspezifikation der Fehlerbehebung von XTP 4.0 202
6.5.4 Livenessverletzungen bei XTP 203
6.6 Einige Gedanken zur Anwendung des kompositionalen Stils 203

7 Schlußbetrachtungen **205**

Kapitel 1

Einleitung

Aus einer Studie der Firma IBM (vgl. [Gib94]), in der 24 große, von führenden Unternehmen entwickelte verteilte Systeme untersucht wurden, geht hervor, daß konventionelle Systementwicklungsmethoden für den Entwurf moderner verteilter Systeme nicht mehr ausreichen. 55 % der untersuchten Systeme überschritten das geplante Kostenbudget und 68 % das Zeitbudget. Sogar 88 % aller Systeme konnten, wenn überhaupt, erst nach substantieller Überarbeitung in Dienst gestellt werden. Zudem sind viele dieser Systeme im Betrieb unzuverlässig. Ein wesentlicher Grund für diese kritische Situation liegt in der hohen Komplexität der Kommunikationsprotokolle, die die Kooperation zwischen den räumlich verteilten Stationen eines Rechnernetzes realisieren.

Ein Ansatz zur Reduktion der Fehleranzahl bei der Entwicklung von Kommunikationsprotokollen liegt in der Anwendung formaler Entwurfstechniken. Durch die formale Spezifikation in einer wohldefinierten Syntax und Semantik erhält man eine präzise Beschreibung des von einem Protokoll geforderten Ablaufverhaltens. Dadurch werden nicht nur Implementierungsfehler aufgrund von Fehlinterpretationen der Beschreibung ausgeschlossen. Formale Spezifikationen ermöglichen vielmehr die Überprüfung der Korrektheit eines Protokollentwurfs durch formale Verifikation. Wenn sowohl die abstrakte Anforderungsdefinition eines Protokolls als auch das implementierbare Programm formal spezifiziert sind, kann man formal verifizieren, daß jedes von der Programmspezifikation beschriebene Ablaufverhalten einem in der Anforderungsdefinition spezifizierten Ablaufverhalten entspricht.

Im Bereich der theoretischen Informatik wurde eine Vielzahl an Methoden zur formalen Spezifikation und Verifikation von Kommunikationsprotokollen entwickelt (Übersichten vgl. [LL90, Win90, HK92]). Als Spezifikationssprachen seien hier Automatenmodelle (z.B. erweiterte Mealy-Automaten [HU79] und Petrinetze [Mer76, Bil91]), kommunizierende sequentielle Prozeßsysteme (z.B. CCS [Mil80]) und logische Sprachen (insbesondere temporale Logiken; vgl. [Lam80, Pnu77]) genannt. Typische Methoden zur formalen Verifikation sind endliche Erreichbarkeitsanalyse [Wes78, ZWR+80], Modell Checking [CES83, ACD91], auf Prozeßalgebren basierende Techniken [Mil80] und logische Deduktion [OG76, AO91b, MP92a, Lam94].

Ein krasser Gegensatz zu der Verfügbarkeit formaler Methoden besteht in der mangelnden Akzeptanz dieser Methoden in der praktischen Entwicklung von Kommunikationsprotokollen. Einige Protokolle werden zwar formal spezifiziert[1]. Die wenigsten Protokolle werden jedoch

[1] Das Normungsgremium ITU der Vereinigung der Post- und Telekommunikationsgesellschaften verlangt

formal verifiziert. Die Ursache dafür liegt in den praktischen Grenzen der Verifikationstechniken. Vollständig automatisierbare Techniken (z.B. Erreichbarkeitsanalyse, Modell Checking) haben trotz diverser Optimierungen (vgl. [Hol88, SLU89, YK90, LM91, BCM+92]) einen sehr hohen Laufzeit- und Speicherplatzbedarf, so daß man mit ihnen die meisten realen Kommunikationsprotokolle nicht überprüfen kann. Interaktive Techniken (z.B. Prozeßalgebren, logische Deduktion) erfordern dagegen einen hohen Aufwand an Zeit. Ihr Einsatz führt deshalb häufig zu einer signifikanten Erhöhung der Entwicklungskosten, die nicht immer durch die dadurch eingesparten Wartungskosten ausgeglichen wird.

Bevor wir den in dieser Arbeit vorgestellten Ansatz beschreiben, mit dem man unseres Erachtens die Akzeptanz formaler Methoden erhöhen kann, stellen wir einige Eigenschaften moderner Kommunikationsprotokolle vor. Durch die Einführung von Hochleistungsrechnernetzen mit einer Datenübertragungsrate von bis zu 2 GBit/s steigen die Anforderungen an die Leistungsfähigkeit der Protokolle stark an, da sich der Leistungsengpaß von der Übertragungsstrecke auf die Protokollinstanzen verlagert, durch die ein Protokoll realisiert wird (vgl. [CT90, LS91a]). Zur Steigerung der Leistungsfähigkeit der Instanzen wurde eine Reihe von Lösungsansätzen erarbeitet, die zu einer Anzahl moderner Hochleistungsprotokolle führte. Ein wesentlicher Gesichtspunkt der meisten Ansätze ist Ausnutzung der Modularität von Protokollen. Jedes Protokoll realisiert unterschiedliche Protokollfunktionen (z.B. Erkennung und Behebung von Übertragungsfehlern, Flußkontrolle, Verbindungsverwaltung), die durch spezielle Protokollmechanismen in den Instanzen erbracht werden. Eine voneinander möglichst unabhängige Gestaltung dieser Protokollmechanismen erleichtert die Realisierung der Instanzen auf leistungsfähigen Hardwareplattformen (vgl. [BZ93, Fel93, GKWW89, Gre91, OP91, Ten89, Zit91]). Man kann zum Beispiel die Mechanismen mit dem höchsten Leistungsbedarf auf speziellen Hardwarebausteinen realisieren. Außerdem kann man Protokolle statisch oder dynamisch konfigurieren, so daß in einer bestimmten Verbindung immer nur Protokollfunktionen aktiv sind, die zur Erbringung der vom Nutzer des Kommunikationssystems geforderten Übertragungsgüte zwingend notwendig sind (vgl. [AP93, CT90, Haa91, LS93, OP92, Tsc91, ZST93]). Eine recht komplexe Protokollklasse bilden die Hochleistungstransportprotokolle, die dem Nutzer eines Kommunikationssystems einen von den technischen Gegebenheiten der verwendeten Rechnernetze unabhängigen Zugang zur Verfügung stellen. Die wesentliche Aufgabe der Instanzen dieser Protokolle besteht dabei in der Überbrückung der Diskrepanz zwischen der vom Nutzer geforderten und der von den Rechnernetzen zur Verfügung gestellten Übertragungsgüten.

Das Ziel dieser Arbeit ist die Entwicklung einer Methode zum formal basierten Entwurf von Hochleistungstransportprotokollen, die gegenüber den herkömmlichen Techniken den Aufwand der Erstellung formaler Spezifikationen und der Durchführung formaler Verifikationen senkt. Als grundlegende formale Spezifikations- und Verifikationstechnik verwenden wir Leslie Lamports Temporal Logic of Actions (TLA) [Lam94]. TLA ist eine einfache temporale Logik, die nur jeweils zwei modale Operatoren und zwei daraus abgeleitete Operatoren verwendet. Das Verhalten eines Systems wird durch eine Menge unendlich langer Zustandsfolgen beschrieben, deren Zustände durch Tupel von Variablenwerten modelliert werden. Die Zustandsübergänge werden durch Aktionen — transitionsähnliche Prädikate über Zustandspaaren — modelliert, so daß man die Systeme durch Zustands-/Transitionssysteme spezifizieren kann. Außerdem erlaubt TLA die explizite Beschreibung von Systemfortschritt durch

die formale Spezifikation aller von ihren Mitgliedsgesellschaften eingesetzten Kommunikationsprotokolle in der Spezifikationssprache SDL [ITU93].

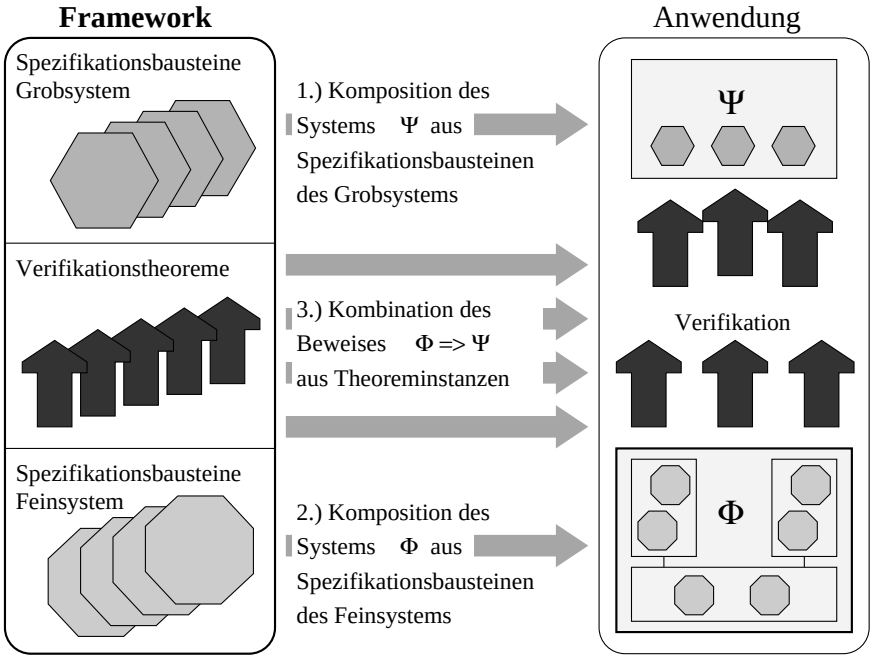

Abbildung 1.1: Aufbau eines Frameworks zur formal basierten Systementwicklung

Angabe von Anforderungen, Aktionen fair auszuführen. Verifikationen in TLA werden mit Hilfe von 20 temporalen Ableitungsregeln geführt. Da die Auswahl dieser Regeln recht schematisch erfolgt, ist die Strukturierung der Beweise meist einfach. Zudem kann man Verfeinerungsbeweise auch durch Angabe von Zustandsabbildungen mit vier Zusatzeigenschaften (Refinement Mappings [AL91a]) erbringen.

Ein wichtiger Ansatzpunkt unserer Methode ist der Einsatz der vor allem aus der Programmiertechnik bekannten Prinzipien der Modularität, Kompositionalität und Wiederverwendbarkeit bei der Spezifikation und Verifikation von Hochleistungsprotokollen. Der modulare Aufbau von Protokollen aus Protokollmechanismen wird ausgenutzt, indem man jeden Mechanismus separat spezifiziert und mit anderen Mechanismen zu einer Protokollspezifikation komponiert (vgl. [AL93a, BLB93, LS91b, MP92b, MC81]). Außerdem werden Verifikationen in Theoreme zerlegt, in denen jeweils nur ein Teil der Protokollmechanismen verifiziert wird. Da die Protokollmechanismen in unterschiedlichen Protokollen eingesetzt werden, sind ihre Spezifikationen so ausgelegt, daß man sowohl sie als auch die sie verifizierenden Theoreme wiederverwenden kann.

Diese Methoden bilden die Grundlage für ein Framework zum Entwurf von Hochleistungsprotokollen, das in dieser Arbeit vorgestellt wird (siehe Abbildung 1.1). Es erlaubt die Spezifikation von Kommunikationsprotokollen und Kommunikationsdiensten, abstrakten Beschreibungen der Schnittstelle zwischen einem Kommunikationssystem und seinen Nutzern, durch Anpassung und Komposition bereits vorhandener Spezifikationsbausteine. Die in Hochleistungstransportprotokollen eingesetzten Protokollmechanismen werden in jeweils einem Spezifikationsbaustein beschrieben. Dabei verzichtet man auf die Festlegung protokoll-

spezifischer Datenstrukturen, sondern beschreibt die Datenstrukturen durch Parameter. Der Protokollentwickler spezifiziert einen Protokollmechanismus eines bestimmten Protokolls, indem er die Parameter eines geeigneten Spezifikationsbausteins instantiiert. Die Bausteine werden miteinander zu einer Protokollspezifikation komponiert. Auf die gleiche Weise wird eine Dienstspezifikation aus Spezifikationsbausteinen komponiert, die die funktionalen Eigenschaften eines Kommunikationsdienstes beschreiben. Das Framework enthält je eine Bibliothek für die Spezifikationsbausteine der Diensteigenschaften und der Protokollmechanismen. Die Bibliothek der Protokollmechanismen umfaßt außerdem Bausteine, die die Eigenschaften des Kommunikationssubsystems modellieren, über das die Protokollinstanzen miteinander kommunizieren.

Das Framework erleichtert auch die formale Verifikation, daß ein Kommunikationsprotokoll einen Kommunikationsdienst erbringt. Eine dritte Bibliothek enthält Theoreme, deren Gültigkeit von uns a priori bewiesen wurde. Jedes Theorem sagt aus, daß eine Spezifikation einer Diensteigenschaft durch ein aus Spezifikationen von Protokollmechanismen komponiertes Subsystem implementiert wird. Der Protokollentwickler untersucht, ob für jede Diensteigenschaft des von ihm spezifizierten Dienstes in der Bibliothek ein Theorem enthalten ist, das sicherstellt, daß die Diensteigenschaft durch ein Subsystem der Protokollspezifikation erbracht wird. Er muß dazu lediglich einige einfache Überprüfungen von Randbedingungen der Theoreme vornehmen. Zum einen muß die Instantiierung der Bausteinparameter konsistent sein. Zum anderen müssen alle in der Protokollspezifikation enthaltenen Bausteine der Protokollmechanismen zueinander verträglich sein. Außerdem muß der Entwickler sicherstellen, daß die Kopplung der Diensteigenschaften und diejenige der Protokolleigenschaften zueinander konsistent sind. Dieser Beweis ist aufgrund der Auslegung der Spezifikationsbausteine trivial.

Bei der Entwicklung des Frameworks mußten einige Probleme gelöst werden. Zum einen mußte das Themengebiet der Hochleistungstransportprotokolle und der Transportdienste daraufhin untersucht werden, geeignete Protokollmechanismen und Diensteigenschaften zu identifizieren, die ein hohes Maß an Modularität und Wiederverwendbarkeit garantieren. Außerdem mußte ein geeigneter Spezifikationsstil entwickelt werden, der sich für kompositionale Spezifikationen eignet. Die Spezifikationstechnik TLA bildet die Basis dieses Stils. Sie eignet sich gut zur Beschreibung einzelner Bausteine. Jedoch erfüllt die von Abadi und Lamport vorgeschlagene Kompositionstechnik für TLA-Spezifikationen [AL93a, AL93b] unsere Anforderungen nicht. Da sie auf geteilten Variablen beruht, läßt sich die Komposition von Spezifikationen, die zueinander synchron ablaufende Protokollmechanismen modellieren, nur unzureichend darstellen. Deshalb mußten wir ein Kopplungsprinzip entwickeln, das zwar auf TLA aufsetzt, jedoch wie CCS [Mil80] und LOTOS [ISO89b] auf simultan ausgeführten Zustandsübergängen in unterschiedlichen Teilspezifikationen basiert. Mit Hilfe dieses Kopplungsprinzips kann man eine sehr spezielle Form von Kompositionalität gewährleisten. Eine Eigenschaft, die von einem Spezifikationsbaustein erfüllt wird, wird zugleich auch von einem System erfüllt, das diesen Baustein als Komponente enthält. Dazu darf das System allerdings keine inneren Widersprüche aufweisen, die sich aus der Kopplung von zueinander inkompatiblen Spezifikationsbausteinen ergeben könnten. Das Kopplungsprinzip schließt derartige Widersprüche aus. Durch diese Form der Kompositionalität ist es möglich, wiederverwendbare Theoreme aufzustellen, mit denen man formale TLA-Beweise in mehrere Teilbeweise zerlegen kann.

Die zugehörige Verifikationstechnik muß die Zerlegung formaler TLA-Verifikationen in mehrere Teilbeweise ermöglichen, in denen jeweils nur Teilspezifikationen des Protokolls und

des Dienstes berücksichtigt werden. Die Teilbeweise werden unter Verwendung der wiederverwendbaren Theoreme erbracht. Daneben ist noch ein Konsistenzbeweis notwendig, der jedoch in hohem Maß aus den Teilergebnissen der Verfeinerungsteilbeweise profitiert und im Normalfall trivial ist. Desweiteren mußte eine gut lesbare und das intuitive Verständnis von Spezifikationen fördernde Spezifikationssprache entwickelt werden. Die von Lamport vorgeschlagene Spezifikationssprache TLA$^+$ [Lam95] erfüllt diese Bedingungen nicht, da Lamport auf syntaktische Konstrukte, die die Lesbarkeit erhöhen, weitgehend verzichtete, um TLA$^+$ möglichst mächtig zu gestalten. Wir setzen deshalb eine von TLA$^+$ abgeleitete Syntax cTLA$^+$ [Hey95] ein, die das Erstellen prozeßorientierter Spezifikationen unterstützt. Außerdem offeriert diese Sprache Konstrukte zur Beschreibung von Operationen auf Spezifikationen, mit denen man Teilspezifikationen an reale Systeme anpassen oder zu einem Gesamtsystem komponieren kann.

Der praktische Wert des Frameworks mußte selbstverständlich untersucht werden. Dazu werden die Monocastfunktionen des Hochleistungstransportprotokolls XTP [Pro89, XTP95], das alle wichtigen Eigenschaften moderner Transportprotokolle besitzt, und mit hoher Wahrscheinlichkeit in der Praxis eingesetzt werden wird, spezifiziert und verifiziert. Dabei stellte sich heraus, daß mit einer kleinen Ausnahme, die jedoch einfach hinzugefügt werden konnte, alle für die Erstellung der Spezifikation notwendigen funktionalen Protokollmechanismen im Framework enthalten sind. Die Gesamtdauer für die Durchführung der Spezifikation und der Verifikation betrug 18 Tage. Diese Dauer und der notwendige Arbeitsaufwand erscheint uns für ein Protokoll dieser Größenordnung bemerkenswert. Bei der Untersuchung wurden einige Ungenauigkeiten in der Protokollbeschreibung von XTP gefunden, die zu Implementierungsfehlern führen können.

Der nachfolgende Text gliedert sich in die folgenden Kapitel: Die Grundlagen der Hochleistungstransportprotokolle werden in Kapitel 2 eingeführt. Kapitel 3 gibt zunächst einen Überblick über Methoden zur funktionalen Modellierung und Verifikation verteilter Systeme. Anschließend wird dort ein Überblick über die von uns verwendete Spezifikations- und Verifikationstechnik TLA und die zugehörige Spezifikationssprache TLA$^+$ gegeben. In Kapitel 4 stellen wir unseren kompositionalen Spezifikationsstil vor, der das gewünschte Kopplungsprinzip realisiert. Anschließend erläutern wir die Methode der strukturierten Verifikation und gehen auf die Spezifikationssprache cTLA$^+$ ein. Das Framework für Hochleistungstransportprotokolle wird in Kapitel 5 eingeführt. Zunächst wird dazu sein Aufbau, die verwendeten Abstraktionsstufen sowie die Struktur der Bibliotheken, skizziert. Daraufhin erläutern wir den Aufbau der einzelnen Spezifikationsbausteine und der Theoreme. Danach werden alle Bausteine der verschiedenen Spezifikationsstufen vorgestellt. Das Kapitel schließt mit einem Überblick über die Theoreme. Für einige Theoreme werden die Verfeinerungsbeweise gezeigt. Die Spezifikation und Verifikation des Fallbeispiels XTP wird in Kapitel 6 erläutert. Dazu wird zunächst auf die besonderen Eigenschaften dieses Protokolls eingegangen. Danach wird durch Parametrisierung von Spezifikationsbausteinen aus dem Framework und anschließender Komposition eine Protokollspezifikation von XTP erzeugt. Mit Hilfe von Theoremen des Frameworks wird nachgewiesen, welche Bausteine der Diensteigenschaften von der Protokollspezifikation erfüllt werden. Abschließend wird das Protokoll bewertet und dabei die in der Untersuchung erkannten Protokollfehler erläutert. Die Arbeit wird durch einige zusammenfassende und ausblickende Betrachtungen in Kapitel 7 abgeschlossen.

Kapitel 2

Hochleistungsprotokolle

In konventionellen Rechnernetzen bildet die relativ niedrige Schrittgeschwindigkeit der Übertragungsmedien den Leistungsengpaß. Durch die Einführung von Lichtwellenleitern werden jedoch Übertragungsgeschwindigkeiten im Bereich über 1 TBit/s zumindestens in der Theorie möglich (vgl. [AK89]), und Rechnernetze mit Übertragungsraten von mehr als 1 GBit/s sind bereits in der Entwicklung. Da sich die Rechnerleistung nicht in der gleichen Geschwindigkeit wie die Leistung der Übertragungsmedien entwickelt hat, verlagert sich der Leistungsengpaß immer mehr von den Übertragungsmedien auf die Rechnernetzknoten und insbesondere auf die Kommunikationsprotokolle (vgl. [CT90]). Die Einführung der Lichtwellenleiter hat jedoch nicht nur Auswirkungen auf die Übertragungsrate, sondern auch auf die Bitfehlerrate, die gegenüber Kupferadern von 10^{-4} auf 10^{-9} sinkt (vgl. [LS91a]). Eine so geringe Bitfehlerrate reicht jedoch bereits für viele Anwendungen wie Audio- oder Videoübertragung aus, so daß die Fehlerbehebungsroutinen gegenüber den konventionellen Protokollen vereinfacht werden können.

Die Einführung von Lichtwellenleitern führt also zu stark veränderten Randbedingungen, so daß eine vollständige Neubewertung der Protokollarchitekturen und der Protokolle selbst notwendig wird. Eine Reihe neuartiger Ansätze wurde veröffentlicht, deren Blick sich insbesondere auf die Aufhebung der durch das ISO/OSI-Basisreferenzmodell [ISO94] vorgegebenen strengen Schichtung, auf die Zerlegung der Protokolle in kleinere Einheiten [CT90, Haa91, LS91a, OP91, Zit91, AP93] und auf die flexible, teilweise dynamische, Anpassung der Protokolle an die aktuellen Protokollanforderungen [LS93, OP92, PPVW93, Tsc91, ZST93] richtet.

In diesem Kapitel sollen die verschiedenen Ansätze und die daraus resultierenden Hochleistungsprotokollrealisierungen vorgestellt werden. In Abschnitt 2.1 gehen wir zunächst auf die Grundlagen der Kommunikationsdienste und -protokolle ein. In Abschnitt 2.2 werden die besonderen Eigenschaften von Hochleistungsrechnernetzen und die daraus resultierenden Anforderungen an die Transportprotokolle diskutiert. Insbesondere werden dabei die Funktionen der Hochleistungsprotokolle vorgestellt. Die oben schon genannten Ansätze werden in Abschnitt 2.3 erörtert. Einige interessante Realisierungen von Hochleistungsprotokollen werden in Abschnitt 2.4 vorgestellt. Abschließend betrachten wir in Abschnitt 2.5, wie man die spezifischen Randbedingungen der Hochleistungskommunikation bei der Entwicklung formaler Modellierungs- und Überprüfungsmethoden berücksichtigen kann.

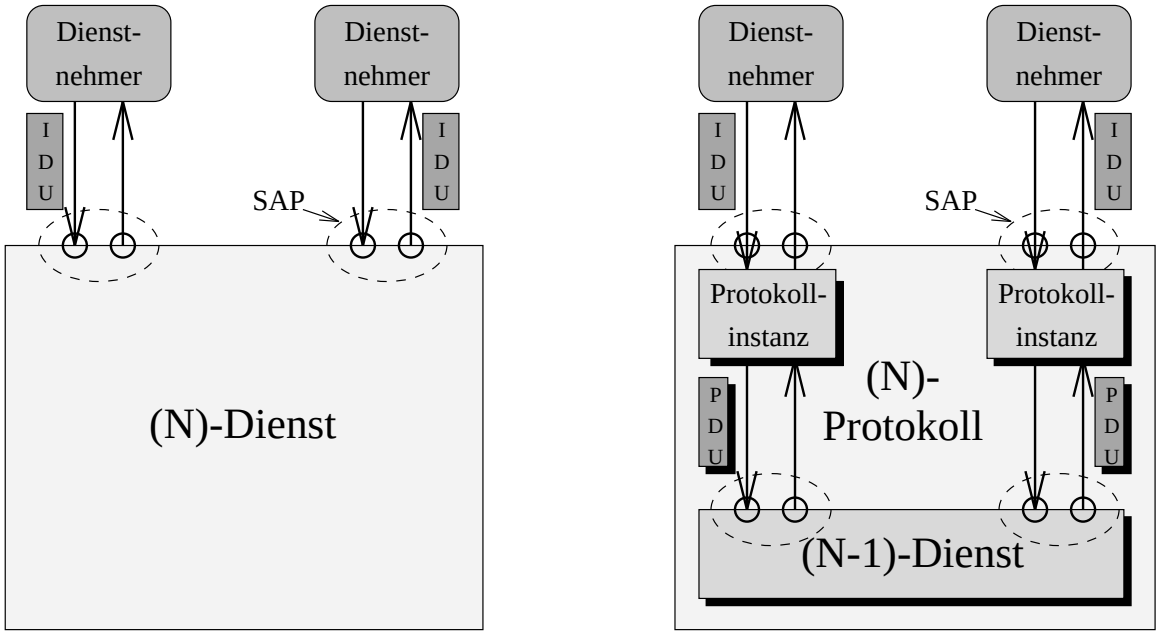

Abbildung 2.1: Kommunikationsdienst und -protokoll

2.1 Grundlagen

Im folgenden soll kurz auf die für das Verständnis der vorliegenden Arbeit wesentlichen Begriffe eingegangen werden. Eine genauere Einführung findet sich in der Grundlagenliteratur über Rechnernetze und verteilte Systeme (z.B. [Hal92, LKK93, SHP91, Tan89, Zit95]). Um räumlich verteilten, zueinander meist heterogenen Anwendungsprozessen die Möglichkeit zu geben, unter Verwendung eines Kommunikationssubsystems miteinander zu kooperieren, muß eine wohldefinierte funktionale Schnittstelle zwischen den Anwendungsprozessen und dem Kommunikationssubsystem gegeben sein. Unter einem Kommunikationsdienst verstehen wir einen Satz an Funktionen, die das Kommunikationssubsystem den Anwendungsprozessen zum Zweck der Datenkommunikation zur Verfügung stellt (siehe linke Seite in Abbildung 2.1). Typische Beispiele für Kommunikationsdienste sind Verbindungsaufbau-, Verbindungsabbau- und Datentransferdienste. Die Schnittstelle zwischen Kommunikationssubsystem und Anwendungsprozeß, die wir im folgenden als Dienstschnittstelle bezeichnen, besteht aus einer Menge einzelner Kommunikationsdienste. In diesem Zusammenhang bezeichnen wir das Kommunikationssubsystem als Diensterbringer und den Anwendungsprozeß als Dienstnehmer. Einzelne Operationen an den Dienstschnittstellen nennen wir Dienstprimitive. Eine Dienstschnittstelle bildet den Zugang für viele räumlich verteilte Dienstnehmer. Der Teil der Schnittstelle, der die Dienstprimitive für genau einen Dienstnehmer enthält, wird als Dienstzugangspunkt (Service Access Point, SAP) bezeichnet. Die zwischen Dienstnehmern und Diensterbringer übertragenen Daten nennen wir Schnittstellendateneinheiten (Interface Data Units, IDUs). Sie bestehen aus Nutzdaten, die für die Kooperation mit anderen Dienstnehmern bestimmt sind (Service Data Units, SDUs), und Schnittstellensteuerinformation (Interface Control Information, ICI), durch die die Kooperation zwischen Dienstnehmer und Diensterbringer

erfolgt (vgl. [Tan89]). Ein wesentliches Unterscheidungskriterium für Dienste ist zum einen die erbrachte Dienstgüte. [LKK93] nennen hier Angemessenheit, technische Leistung, Kosten, Zuverlässigkeit und Schutz als Hauptaspekte. Zum anderen kann man Dienste auch hinsichtlich der durch sie realisierten Funktionen, der Diensteigenschaften, unterscheiden. Typische funktionale Eigenschaften sind Phantomerkennung oder der Ausschluß von Datenverfälschungen.

Außer Vereinbarungen zur Kooperation zwischen Dienstnehmer und Diensterbringer sind auch Regeln zur Kommunikation von zwei oder mehr zueinander räumlich entfernten Dienstnehmern notwendig. Einen solchen Regelsatz nennt man Kommunikationsprotokoll (siehe rechte Seite in Abbildung 2.1). Durch Kommunikationsprotokolle werden zum einen der Ablauf der Kooperation zwischen den Dienstnehmern und zum anderen das Format der übertragenen Daten festgelegt. Die miteinander nach den Protokollregeln kooperierenden Komponenten nennt man Protokollinstanzen. Die zwischen ihnen übertragenen Daten werden als Protokolldateneinheiten (Protocol Data Units, PDUs) bezeichnet. Sie bestehen aus der Protokollsteuerinformation (Protocol Control Information, PCI), die die für die Kooperation notwendigen Steuerungsdaten enthält, und den zu übertragenden SDUs. Wie die Dienste kann man auch die Protokolle in funktionale Einheiten zerlegen, die als Protokollfunktionen bezeichnet werden. Fehlererkennung, Datenwiederholung und Flußkontrolle sind typische Beispiele für Protokollfunktionen. Die Einheiten einer Protokollinstanz, die zur Realisierung bestimmter Protokollfunktionen eingesetzt werden, nennt man Protokollmechanismen.

Die Protokollinstanzen können mit Hilfe des Protokolls und des zur Übertragung der PDUs genutzten Dienstes ihrerseits einen Dienst höherer Qualität erbringen. Damit ist eine funktionale Schichtung des Kommunikationssubsystems möglich. Eine Schicht der Kategorie (N) besteht aus einem (N)-Dienst und Komponenten, die ihn realisieren. Das sind (N)-Protokollinstanzen und ein $(N-1)$-Dienst, über den die Protokollinstanzen kooperieren. Dabei werden die vom (N)-Dienstnehmer übergebenen SDUs von den (N)-Protokollinstanzen als Nutzdaten in (N)-PDUs unter Verwendung des $(N-1)$-Dienstes übertragen. [OP92] bezeichnen eine aus mehreren Schichten bestehende hierarchische Protokollstruktur als Protokollgraph. Ein Protokoll in einer Schicht erbringt immer nur einen Teil der Funktionalität des gesamten Kommunikationssystems.

In Rechnernetz-Standards werden Referenzmodelle für Kommunikationsarchitekturen verwendet, in denen Kommunikationssysteme in Schichten zerlegt sind. Erwähnt seien das DoD-Referenzmodell [Pad83], das aus vier Schichten besteht, und insbesondere das ISO/OSI-Basisreferenzmodell [ISO94], das von der Arbeitsgruppe „Open Systems Interconnection“ der Internationalen Standardisierungsorganisation ISO (International Organzation for Standardization) genormt wurde. Wir beschreiben im folgenden kurz seine sieben Schichten. Die Realisierung der Datenübertragung über ein physikalisches Medium, insbesondere die Kodierung und Dekodierung der physikalischen Signale obliegt den Protokollmechanismen der Bitübertragungsschicht ((1)-Schicht). Die Protokolle der Sicherungsschicht ((2)-Schicht) dienen der Sicherstellung des ordnungsgemäßen Datentransfers zwischen zwei durch ein physikalisches Medium verbundener Stationen. Dazu gehören Mechanismen zur Datensicherung, zur Flußkontrolle und zur Sicherstellung eines exklusiven Zugriffs bei Kanälen mit wahlfreiem Zugriff. Die Sicherstellung des ordnungsgemäßen Datentransfers zwischen zwei Endsystemen ist Aufgabe der Vermittlungsschicht ((3)-Schicht). Dabei sind die Aspekte der Vermittlung von Ende-zu-Ende-Verbindungen aus einer Folge verschiedener physikalischer Kanäle wesentlich, die über Vermittlungsknoten verbunden sind. Die Qualität der Dienste der genannten

Schichten hängt direkt von technischen Aspekten des verwendeten Rechnernetzes ab. Nutzer eines Kommunikationssubsystems wünschen jedoch eine Dienstgüte, die nur von den Anwendungsbedürfnissen abhängt. Aufgabe der (4)-Protokollinstanzen ist die Realisierung derartiger Dienstgüten auf der Basis der existierenden Rechnernetze. Je nach Qualität der (3)-Dienste müssen die (4)-Protokollinstanzen auch Protokollmechanismen realisieren, die eigentlich den niedrigeren Schichten zugeordnet sind. Die bisher skizzierten Schichten werden auch als transportorientierte Schichten bezeichnet, weil sie die technischen Aspekte des Kommunikationssystems abbilden. Das System, das durch diese vier Schichten gebildet wird, nennt man Transportsystem. Die Aufgabe der drei oberen Schichten des ISO/OSI-Basisreferenzmodells ist dagegen eine direkte Unterstützung der Kommunikationsanwendungen. Die Protokolle der Kommunikationssteuerungsschicht ((5)-Schicht) stellen Dienste zur transaktionsorientierten Sicherung des Datenstroms gegen Störungen und Ausfälle des Transportsystems zur Verfügung. Um die Kommunikation von zueinander heterogenen Anwendungsprozessen mit unterschiedlichen internen Datendarstellungsformaten zu ermöglichen, wird für die Übertragung eine für alle Kooperationspartner eindeutige Transfersyntax verwendet. Die Festlegung von Transfersyntaxen obliegt den Diensten der Darstellungsschicht ((6)-Schicht). Die Aufgabe der Anwendungsschicht ((7)-Schicht), die von [LKK93] treffender Anwendungskommunikationsschicht genannt wird, ist die Realisierung von Anwendungsdienstelementen, die jeweils einen speziellen anwendungsnahen Kommunikationsdienst erbringen. Typische Anwendungsdienstelemente sind RTSE (Reliable Transfer Service Element), das einen zuverlässigen Datentransferdienst sicherstellt, und CCR (Commitment, Concurrency, and Recovery), das die Atomizität bei verteilten Transaktionen unterstützt.

2.2 Transportprotokolle

Wie schon kurz ausgeführt wurde, besteht die Aufgabe der Transportprotokolle in der Abbildung von durch Transportdienstnehmer angebotenen Diensten auf Netzwerkdienste. Im folgenden beschränken wir uns im wesentlichen auf verbindungsorientierte Transportprotokolle. Mit Ausnahme von UDP [Pos80] und dem verbindungslosen ISO/OSI-Transportprotokoll [ISO86b] sind alle uns bekannten Transportprotokolle verbindungsorientiert, weil wesentliche Anwendungen wie Bewegtbildübertragung nur durch Verbindungen sinnvoll unterstützt werden können (vgl. [PT90]). Allerdings bieten einige Protokolle sowohl verbindungsorientierten Datentransfer als auch verbindungslosen Datentransfer an, der durch nur für kurze Zeit aufgebaute Transportverbindungen realisiert wird (vgl. [Bra93, SW90]). Im folgenden skizzieren wir zunächst kurz den Transportdienst. Aufgabe der Instanzen des Transportprotokolls ist es, die vom Dienstnehmer geforderten Dienstgüteanforderungen sinnvoll auf Netzwerkverbindungen abzubilden. Gegebenenfalls muß die Qualität der Netzwerkverbindungen durch die Verwendung zusätzlicher Protokollmechanismen weiter verbessert werden, die nach der ISO/OSI-Architektur eigentlich durch die Protokollinstanzen der Sicherungsschicht und Vermittlungsschicht erbracht werden sollen.

Aufgrund der unter Umständen starken Diskrepanz der Dienstgüten zwischen Transportverbindungen und Netzwerkverbindungen existiert eine Vielzahl an Protokollfunktionen für Transportprotokolle (vgl. [DDK+90, LS91a, Svo89]). Aufgaben der Protokollfunktionen sind zum einen die Steuerung des Nutzdatentransfers der Transportverbindung über Netzwerkverbindungen und zum anderen die Verwaltung der Transportverbindung. Eigenschaften der Da-

tentransfersteuerung sind die Behandlung von Übertragungsfehlern, die Bestätigung erfolgreich übertragener Dateneinheiten und die Flußkontrolle. Aspekte der Verbindungsverwaltung sind Verbindungsaufbau und -abbau, Übertragung von Zustandsinformation sowie Multiplexing. Diese Protokollfunktionen werden in den Abschnitten 2.2.2 und 2.2.3 beschrieben. Reale Protokollsysteme unterscheiden sich voneinander nicht nur hinsichtlich ihrer Funktionalität und der erbrachten Dienstgüte sondern auch durch die verwendete Systemarchitektur. Wir stellen in Abschnitt 2.2.4 ein Architekturmodell für Transportsysteme vor.

2.2.1 Der Transportdienst

Da die Hauptaufgabe des Transportsystems in der Realisierung von Verbindungen bestimmter anwendungsabhängiger Dienstgüten liegt, spielt die Dienstgütebestimmung bei der Festlegung der Transportdienstschnittstelle eine zentrale Rolle. Der ISO/OSI-Transportdienst [ISO86c] sieht einen bestätigten Verbindungsaufbaudienst (*T-CONNECT*) und unbestätigte Dienste für Verbindungsabbau (*T-DISCONNECT*), Datentransfer (*T-DATA*), Datentransfer für Vorrangdaten (*T-EXPEDITED-DATA*) und verbindungslosen Datentransfer (*T-UNIT-DATA*) vor. Da andere Protokolle zum Teil abweichende Verfahren zum Verbindungsaufbau und -abbau verwenden (siehe Abschnitt 2.2.3), unterscheiden sich ihre Dienstschnittstellen in den Dienstprimitiven.

Beim Aufbau einer Transportverbindung handeln die Transportdienstnehmer und das Transportsystem eine bestimmte Dienstgüte (Quality of Service, QoS) der Verbindung aus. Dabei werden für die verschiedenen Verbindungsphasen und die Gesamtverbindung Toleranzen qualitativer Parameter angegeben (vgl. [SHP91]). Der Verbindungsaufbau und der Verbindungsabbau werden durch seine maximale Dauer und durch die Wahrscheinlichkeit eines Dienstabbruchs aufgrund eines Systemausfalls charakterisiert. Kenngrößen für die Datentransferphase sind Durchsatz, Übertragungsverzögerung und Restfehlerrate. Weiterhin wird festgelegt, mit welcher Wahrscheinlichkeit eine dieser Kenngrößen nicht eingehalten werden darf. Für die Transportverbindung als ganzes können die Priorität der Verbindung gegenüber anderen Verbindungen, die Verwendung von Verschlüsselungsverfahren und eine maximal tolerierte Wahrscheinlichkeit für vorzeitige Verbindungsabbrüche aufgrund von Fehlern im Transportsystem angegeben werden.

2.2.2 Protokollfunktionen des Datentransfers

Übertragungsfehler können verlorengegangene Daten, Phantomdaten, duplizierte Daten, Daten, deren Reihenfolge vertauscht ist, und verfälschte Daten sein (vgl. [LKK93]). Die Erkennung von Phantomen und Verfälschungen erfolgt durch Hinzufügung redundanter Information (Prüfsummen) in PDUs. Dabei wird am häufigsten die sogenannte „Cyclic Redundancy Code“-Methode eingesetzt. Verlorengegangene, duplizierte und vertauschte Nachrichten können durch Sequenznummern erkannt werden. Dabei kann entweder jedem einzelnen Nutzdatenbyte (z.B. Transportprotokolle TCP [Pos81], XTP [Pro89]) oder nur jeder Nutzdateneinheit (z.B. ISO/OSI-TP4 [ISO86a], VMTP [Che88]) eine Sequenznummer zugeordnet werden. Sequenznummern von als fehlerbehaftet erkannten Nutzdaten können ihrem Sender vom Empfänger explizit gemeldet (Selective Reject, z.B. XTP, VMTP) oder einfach ignoriert werden (z.B. TCP, ISO/OSI-TP4). Bitfehler werden durch fehlerkorrigierende Codes oder, wie in fast allen bekannten Transportprotokollen, durch Übertragungswiederholung behoben.

Übertragungswiederholungen können durch positive Bestätigungen (Positive Acknowledgment with Retransmission, PAR) oder durch explizite Zurückweisungen (Automatic Repeat Request, ARQ) initiiert werden. Bei PAR werden Nutzdaten erneut gesendet, wenn sie nicht innerhalb einer bestimmten Zeitspanne bestätigt werden konnten. Ein Zeitgeber kann dabei entweder einer einzigen Dateneinheit (z.B. TCP, ISO/OSI-TP4) oder einem Block von Dateneinheiten (z.B. NETBLT [CLZ87], XTP, SNR [NRS90]) zugeordnet sein. Bei ARQ werden Dateneinheiten wiederholt, wenn sie von ihrem Empfänger explizit zur erneuten Übertragung angefordert werden (z.B. XTP, VMTP). Ein weiteres Merkmal ist die Anzahl der nach einem Fehler erneut zu übertragenden Dateneinheiten. Bei der Rücksetzung um n Dateneinheiten (Go-back-n) müssen alle Dateneinheiten wiederholt werden, die hinter der fehlerbehafteten Dateneinheit gesendet wurden (z.B. XTP). Bei der selektiven Wiederholung (Selective Retransmission) müssen nur die fehlerhaft übertragenen Dateneinheiten selbst erneut gesendet werden (z.B. NETBLT, VMTP). Die Go-back-n-Methode führt zu einer größeren Anzahl zu wiederholender Daten, erlaubt aber eine einfachere Verarbeitung empfangener Daten als die selektive Wiederholung.

Da jederzeit Übertragungsfehler auftreten können, muß der Sender alle Dateneinheiten bis zu ihrem ordnungsgemäßen Empfang zwischenspeichern. Der Empfänger muß also die ordnungsgemäß empfangenen Dateneinheiten gegenüber dem Sender bestätigen. Bestätigungsmitteilungen können zum einen durch den Empfänger als Reaktion auf ein Ereignis des Senders initiiert werden (Sender-dependent Acknowledgements; z.B. TCP, VMTP, NETBLT, ISO/OSI-TP4). In XTP werden Bestätigungen sogar explizit durch den Sender angefordert. Zum anderen können Bestätigungen unabhängig vom Sender durch einen Zeitgeber des Empfängers gesteuert werden (Sender-independent Acknowledgements; z.B. ISO/OSI-TP4). Bestätigt werden entweder alle empfangenen Nachrichten (z.B. XTP) oder nur der Teil des Datenstroms, der ohne Lücken empfangen wurde (z.B. ISO/OSI-TP4). In diesem Fall ist die Übertragung einer einzigen Sequenznummer ausreichend.

Durch die Mechanismen der Flußkontrolle wird sichergestellt, daß die Rate gesendeter Dateneinheiten der Verarbeitungskapazität des untergeordneten Netzwerks bzw. des Empfängers angepaßt ist. Damit kann die Wahrscheinlichkeit für Überschreitungen der Puffergrenzen, woraus zumindestens in Hochleistungsprotokollen die meisten Datenverluste resultieren (vgl. [CW89]), minimiert werden. Flußkontrolle wird entweder durch Kreditvergabeschemata (End-to-end Flow Control) oder durch temporäre Sendebeschränkungen (Rate Control) realisiert. In den Kreditvergabeschemata beschränkt der Empfänger die Anzahl gesendeter Dateneinheiten durch Sendekredite. In Abhängigkeit von seiner Pufferkapazität erlaubt er seinem Partner, nur eine bestimmte Anzahl an Dateneinheiten (Cumulative Window; z.B. NETBLT) oder aller Dateneinheiten, deren Sequenznummer unter einer oberen Grenze liegt (Absolute Window; z.B. ISO/OSI-TP4, TCP, XTP), zu senden. Temporäre Sendebeschränkungen basieren entweder auf durch Zeitgeber gesteuerte Sendepausen zwischen zwei gesendeten Dateneinheiten (Interpacket Delay; z.B. VMTP) oder auf einer datenratenbasierten Sendesteuerung (Burst-size and Interval; z.B. XTP, NETBLT). Bei dieser Methode dürfen Dateneinheiten nur gesendet werden, wenn weder die mittlere Rate aller gesendeten Daten noch die Senderate bei burstartigem Datenaufkommen eine bestimmte Obergrenze überschritten hat. Diese Obergrenzen können durch den Empfänger modifiziert werden. Temporäre Sendebeschränkungen eignen sich auch zur Verhinderung von Überlastungen des Endsystems (Access Control). Dabei kann zum einen das Endsystem die Einhaltung einer maximalen Datenrate verlangen (z.B. XTP). Zum anderen können die Protokollmechanismen selbst anhand der geschätzten

Datenumlaufzeit geeignete Datenraten berechnen (z.B. TCP, VMTP; vgl. auch [Jac88]).

2.2.3 Protokollfunktionen der Verbindungsverwaltung

In Transportprotokollen werden drei unterschiedliche Verfahren zum Verbindungsaufbau und zum ordnungsgemäßen Verbindungsabbau von Transportverbindungen verwendet. Beim impliziten Verbindungsaufbau kann der Initiator unmittelbar nach Sendung einer Verbindungsaufbau-Nachricht mit dem Datentransfer beginnen (z.B. XTP, VMTP). Der Empfänger dieser Nachricht sendet keine explizite Bestätigung, sondern beginnt ebenfalls mit dem Datentransfer. Die erste von ihm gesendete Dateneinheit hat dann die Funktion einer impliziten Bestätigungsnachricht. Transportprotokolle, die Verbindungen implizit abbauen, sind uns nicht bekannt. Einige Protokolle (VMTP, PATROCLOS [Bra93]) unterstützen dennoch unbestätigten Verbindungsabbau. Wenn auf einer Transportverbindung über einen bestimmten Zeitraum keine Daten mehr übertragen wurden, löst eine der Protokollinstanzen von sich aus einen Verbindungsabbau aus. Implizite Verbindungsverwaltungsverfahren sind insbesondere bei kurzlebigen Verbindungen interessant, da sie nur eine geringe Verzögerung durch die Verbindungsverwaltung erfordern.

Bestätigte Verfahren sind dagegen weniger empfindlich gegen Datenverluste. Beim einfach bestätigten Verbindungsaufbau (2-Way Handshake) darf der Initiator erst Daten senden, nachdem der Verbindungsaufbau durch seinen Partner bestätigt wurde (z.B. NETBLT), während der Partner sofort nach Senden der Bestätigung mit dem Datentransfer beginnen darf. Dadurch ist ausgeschlossen, daß vom Initiator gesendete Dateneinheiten durch zu frühzeitiges Ausliefern verloren gehen. Um auch den Partner vor diesem Fehler zu schützen, sendet er beim doppelt bestätigten Verbindungsaufbau (3-Way Handshake) erst Daten, wenn seine Verbindungsaufbaubestätigung durch den Initiator ebenfalls bestätigt wurde (z.B. TCP, ISO/OSI-TP4). Diese zweite Bestätigung kann auch eine Dateneinheit sein.

Doppelte Bestätigungen bieten auch beim ordnungsgemäßen Verbindungsabbau einen höheren Schutz vor Datenverlusten. Bei einfacher Bestätigung (z.B. NETBLT, ISO/OSI-TP4) kann zwar der Verbindungsabbauinitiator, nicht jedoch sein Partner sicherstellen, daß alle gesendeten Dateneinheiten ausgeliefert wurden. Beim Verbindungsabbau mit doppelter Bestätigung (z.B. TCP, XTP, PATROCLOS) werden beide Datentransferrichtungen voneinander unabhängig abgebaut. Der Initiator zeigt durch die Verbindungsabbau-PDU, daß er keine Dateneinheiten mehr senden möchte. Sein Partner darf nach Erhalt dieser PDU jedoch weitersenden. Erst, nachdem der Empfang für alle von ihm gesendeten Dateneinheiten quittiert wurde, bestätigt der Partner den Verbindungsabbau. Der Initiator zeigt ihm den erfolgreichen Abschluß der Verbindung durch eine weitere Bestätigung an.

Außer geordneten Verbindungsabbauverfahren unterstützen alle Transportprotokolle Dienste zum sofortigen, nicht bestätigten Verbindungsabbruch bei Störungen des Dienstnehmers bzw. des Diensterbringers.

Verbindungsaufbau- und Verbindungsabbauinformation, aber auch andere Verbindungsverwaltungs-PDUs wie Datenbestätigungen, Wiederholungsanforderungen oder Kreditvergabenachrichten (siehe Abschnitt 2.2.2) kann man entweder über die gleichen Netzwerkverbindungen wie die Daten-PDUs (In-band; z.B. TCP, ISO/OSI-TP4, XTP) senden oder dafür eigene Netzwerkverbindungen verwenden (Out of Band; z.B. Datakit [Fra83]).

Unter Multiplexing versteht man die Abbildung mehrerer Transportverbindungen auf eine Netzwerkverbindung. Multiplexing wird von XTP, VMTP, ISO/OSI-TP4, TCP und NETBLT

unterstützt, während Datakit und PATROCLOS auf Multiplexing verzichten. Umgekehrt kann auch eine Transportverbindung auf mehrere Netzwerkverbindungen abgebildet werden (Bündeln; z.B. ISO/OSI-TP4).

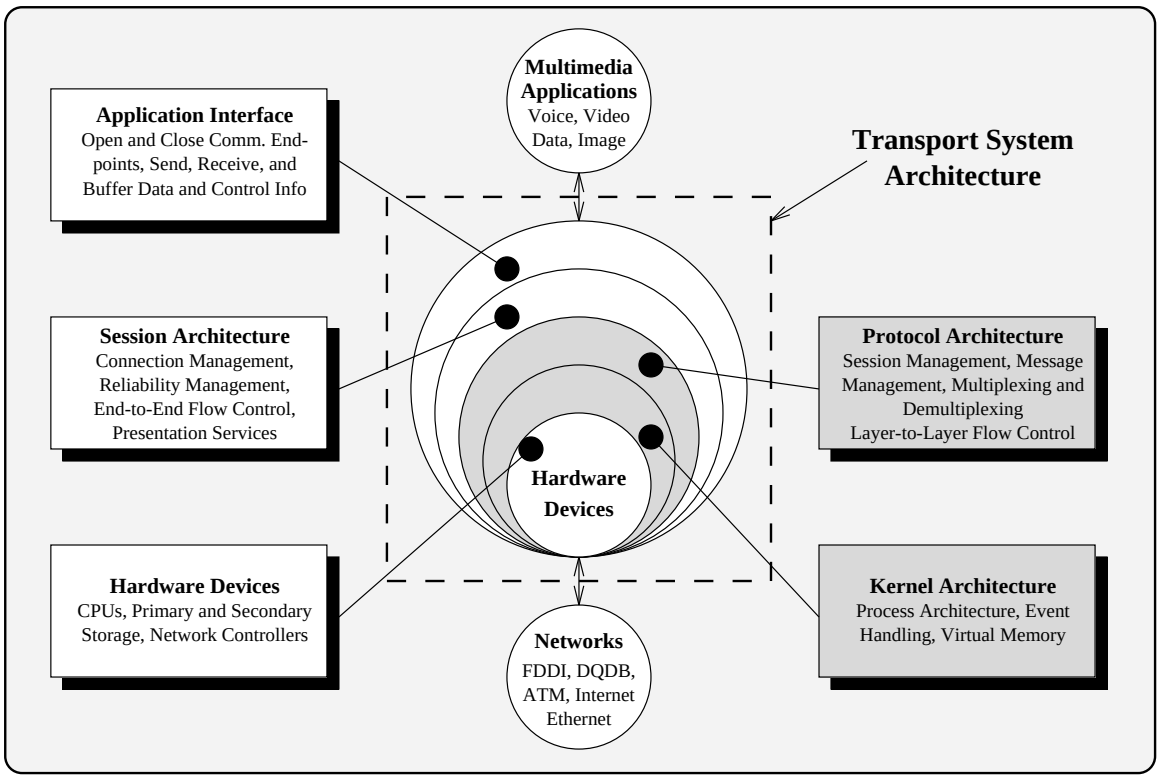

Abbildung 2.2: Transportsystem-Architektur (entnommen aus [SS93])

2.2.4 Transportprotokoll-Architekturen

Schmidt und Suda stellen in [SS93] ein Architekturmodell für Transportsysteme vor, das nicht nur die funktionalen Protokollkomponenten, sondern auch die Schnittstelle zu den beteiligten Hardwareressourcen umfaßt. Das Modell definiert fünf verschiedene Systemebenen, denen die Software- und Hardwarekomponenten eines Protokolls zugeordnet werden (siehe Abbildung 2.2).

Die Anwendungsschnittstelle (Application Interface) beschreibt die Systemschnittstelle, über die Anwendungsprozesse auf die Protokollsoftware zugreifen. In der Protokollterminologie entsprechen diese Schnittstelle dem Dienstzugangspunkt und die Schnittstellenfunktionen den Dienstprimitiven (vgl. [ISO94]).

Der Verbindungsarchitektur (Session Architecture) sind die Protokollkomponenten zugeordnet, die mit den räumlich entfernten Protokollinstanzen kooperieren. Das sind die Protokollmechanismen, die die in den Abschnitten 2.2.2 und 2.2.3 vorgestellten Protokollfunktionen realisieren.

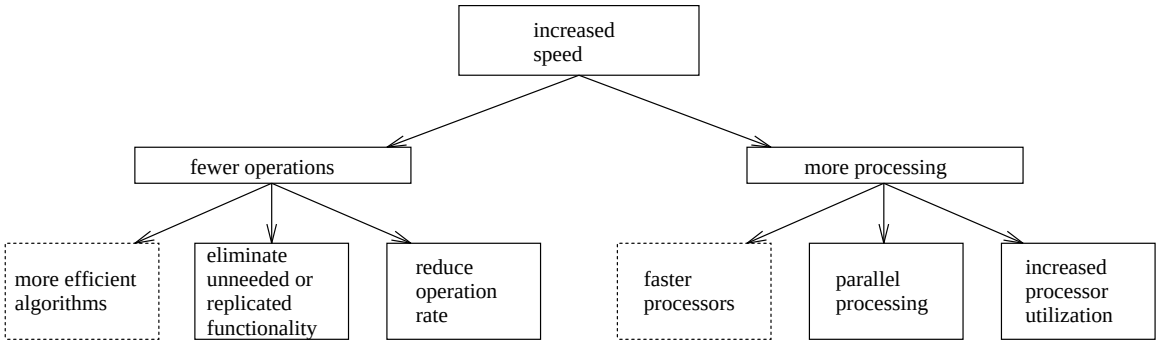

Abbildung 2.3: Methoden zur Beschleunigung von Kommunikationssystemen (entnommen aus [Fel93])

Die Protokollarchitektur (Protocol Architecture) enthält die Protokollkomponenten, die die Interaktion mit der Protokollinstanz der nächsttieferen Schicht im Protokollgraph koordinieren. Dazu gehören die Verwaltung des Dienstzugangspunktes des untergeordneten Dienstes, Datenmanipulationsfunktionen wie die Erzeugung von PDUs und IDUs und auch Mechanismen wie Datensegmentierung, Verschlüsselung oder Multiplexing.

Die Funktionen zum direkten Zugriff auf Betriebssystemressourcen wie CPU, Speicher oder Ein-/Ausgabegeräte sind der Kernarchitektur (Kernel Architecture) zugeordnet.

Die unterste Ebene beschreibt die Hardware, die den Protokollablauf realisiert.

2.3 Hochleistungsprotokollarchitekturen

Die Ankunftszeit eines 1 KByte langen Datenpakets in einem Rechnernetz mit der Übertragungsrate von 1 GBit/s beträgt acht Mikrosekunden. Clark et al. haben berechnet, daß eine Standardimplementierung des Protokolls TCP/IP (Berkley TCP) zum Empfang eines Datenpakets 335 Instruktionen benötigt [CJRS89], so daß also eine Prozessorleistung von ca. 42 MIPS notwendig ist, um alle empfangenen Datenpakete in Echtzeit zu verarbeiten. Das entspricht ungefähr der Rechenleistung einer handelsüblichen Workstation. Standardimplementierungen von konventionellen Transportprotokollen wie TCP oder ISO/OSI-TP können somit nicht unter vertretbaren Kosten für Hochleistungsrechnernetze eingesetzt werden. Ein wesentlicher Grund dafür ist, daß diese Protokolle an relativ langsame, stark fehlerbehaftete Rechnernetze angepaßt sind (vgl. [LS91a]), während sich Hochleistungsnetze durch hohe Übertragungsraten und niedrige Bitfehlerwahrscheinlichkeiten auszeichnen.

Da sich aufgrund dieser Eigenschaften auch das Spektrum von Rechnernetzanwendungen verändert hat, geben wir in Abschnitt 2.3.1 zunächst einen Überblick über typische Anwendungen von Hochleistungsnetzen. In Abschnitt 2.3.2 zeigen wir dann, wie man konventionelle Protokolle optimieren kann. Da man durch derartige Optimierungen die Übertragungsrate jedoch auf nur etwa 100 MBit/s erhöhen kann, ist die Entwicklung neuartiger Transportprotokolle notwendig. Feldmeier [Fel93] sieht zwei grundlegende Ansätze zur Entwicklung von Hochleistungsprotokollsystemen (siehe Abbildung 2.3). Man kann Protokolle entwickeln, die für die Datenverarbeitung weniger Operationen benötigen. Das können effizientere Algorith-

men sein. In Abschnitt 2.3.3 zeigen wir, welche Protokollfunktionen sich für Hochleistungsrechnernetze eignen. Eine weitere Verbesserungsmöglichkeit ist eine Reduktion der Operationen, die durch Veränderungen des Protokollgraphen erreicht werden kann. Lösungsansätze stellen wir in Abschnitt 2.3.4 vor. Eine geringere Anzahl von Operationen kann auch durch Anpassung der Protokollfunktionen an die aktuellen Bedürfnisse der Anwendungen und die Eigenschaften des Rechnernetzes erreicht werden (Abschnitt 2.3.5). Der zweite Ansatz sieht den Einsatz von Plattformen mit einer höheren Rechnerleistung vor. In Abschnitt 2.3.6 zeigen wir Möglichkeiten zur parallelen Verarbeitung von Protokollfunktionen auf.

2.3.1 Anwendungen von Hochleistungsnetzen

Ein wichtiges Anwendungsfeld von Hochleistungsrechnernetzen sind multimediale Systeme. Sie nutzen sowohl die hohe Datenübertragungsrate und die geringe Übertragungsverzögerung bei der Realisierung kontinuierlicher und diskreter Medien (vgl. [DTH92, HS91]). Bei kontinuierlichen Medien wie Bewegtbild- oder Sprachübertragung muß ein isochroner Datenstrom übertragen werden. Der Empfänger toleriert Datenverluste und -verfälschungen von bis zu 2 % der übertragenen Daten, verlangt jedoch einen unterbrechungsfreien Datenstrom (vgl. [Muk91, SHGC92]). Im Gegensatz zur Sprachübertragung, die mit Schmalband-ISDN B-Kanälen von 64 KBit/s auskommt (vgl. [LKK93]), beansprucht die unkomprimierte Übertragung von hochauflösenden Fernsehbildern (High Definition Television, HDTV) eine Datenrate von 1,5 GBit/s und für die komprimierte Übertragung 150 MBit/s (vgl. [Zit95]). Das Transportsystem muß also über einen längeren Zeitraum einen Kanal mit gleichbleibend hoher Datenrate zur Verfügung stellen. Aufgrund der niedrigen Restfehlerrate von Lichtwellenleitern sind dabei außer einer Reihenfolgekontrolle keine Fehlerbehebungsmaßnahmen notwendig.

Textdokumente und Graphiken sind Beispiele für diskrete Medien in multimedialen Systemen, die durch anisochronen Datentransfer übertragen werden. Der Datenstrom kann durchaus für kurze Zeit unterbrochen werden. Allerdings wird eine sehr niedrige Bitfehlerrate und eine kurze Verzögerungszeit verlangt (vgl. [NH89]). Bei multimedialen RPC-Diensten werden nur Reaktionszeiten von weniger als einer Millisekunde toleriert (vgl. [Lid90]). Das Transportsystem muß also eine schnelle und fehlerfreie Datenübertragung realisieren.

Ein anderer zukunftsträchtiger Anwendungsbereich ist die verteilte Verarbeitung, in der anstatt Mehrprozessorsystemen schnelle verteilte Systeme eingesetzt werden (vgl. [BCF93, BvdM93]). Diese Systeme haben ähnliche Anforderungen an das Transportsystem wie die diskreten Medien. Daten müssen mit nur kurzer Verzögerung und ohne Verfälschungen übertragen werden (vgl. [SS94]). Die Anzahl der auf einmal zu übertragenden Daten variiert in Abhängigkeit von der Anwendung stark. Eine besondere Herausforderung ist die medizinische Bildverarbeitung, in der Daten von Röntgenbildern oder Computertomographien, die oft eine Größe von einem GByte erreichen, schnell und mit hoher Präzision übertragen werden müssen (vgl. [RS92]). Viele Anwendungen, in denen jeweils immer nur eine kleine Anzahl an Daten zu übertragen ist, tolerieren Datenvertauschungen (z.B. verteilte Berechnungen). Dafür ist der Einsatz eines verbindungslosen Dienstes sinnvoll (vgl. [LS91a]).

2.3.2 Optimierung konventioneller Protokolle

Untersuchungen haben gezeigt, daß nicht die Protokollfunktionen eines Transportprotokolls, sondern die Datenmanipulation und die Betriebssystemzugriffe entscheidend für die System-

leistung sind. Die Funktionen der Anwendungsschnittstelle im Architekturmodell (siehe Abschnitt 2.2.4) benötigen zwischen 20 % [CJRS89] und 40 % [HP91] der Gesamtleistung. Ausschlaggebend ist dafür die Notwendigkeit, Transferdaten zwischen Anwenderspeicher und Transportsystemspeicher zu kopieren. Die Funktionen des Betriebssystemzugriffs, die der Ebene der Kernarchitektur zugeordnet sind, benötigen noch einmal 25 % der Systemleistung. Clark et al. [CJRS89] folgern daraus, daß man auch konventionelle Protokolle wie TCP durch eine effiziente Implementierung dieser Funktionen für Transportprotolle mit einer Übertragungsrate von mindestens 100 MBit/s betreiben kann. Sie schlagen vor, die PDU-Kopferzeugung und -überprüfung durch Erzeugung von Kopfmustern zu beschleunigen. Der Overhead, der durch die Zuordnung von Verbindungen zu PDUs entsteht, soll durch eine intelligente Suchstrategie abgebaut werden, nach der zuerst die Verbindungen überprüft werden, auf denen zuletzt gesendet wurde. Der für den Betriebssystemzugriff notwendige Aufwand soll durch vollständigen DMA-Einsatz beim Kopieren der Daten und durch optimierte Zeitgebersysteme gemindert werden.

Eine weitere Möglichkeit zur Verbesserung der Leistung von TCP und ISO/OSI-TP ist die Integration der Protokollfunktionen der verschiedenen Protokollschichten (vgl. [AP93, Tsc91]). Dadurch kann die mehrfache Ausführung von Protokollfunktionen in verschiedenen Protokollschichten verhindert werden. Ein Beispiel dafür ist die Flußkontrolle im ISO/OSI-Basisreferenzmodell [ISO94], die als Punkt-zu-Punkt-Flußkontrolle in der Sicherungsschicht und als Ende-zu-Ende-Flußkontrolle in der Vermittlungsschicht durchgeführt wird. Auch zueinander parallel ausgeführte Modifikationen von Nutzdaten wie Verschlüsselung oder Prüfsummenberechnung werden nur durch Aufhebung der strengen Schichtung möglich. Die Integration von Protokollfunktionen wird in Abschnitt 2.3.4 näher erläutert.

2.3.3 Adaption der Protokollfunktionen

In den Abschnitten 2.2.2 und 2.2.3 wurde eine Übersicht über die wesentlichen Protokollfunktionen für den Einsatz in Transportprotokollen gegeben. Einige Aufgaben von Protokollen können durch unterschiedliche Protokollfunktionen realisiert werden. Die Auswahl geeigneter Protokollfunktionen, die die hohe Bandbreite des Übertragungsmediums und den Leistungsengpaß der Netzwerkknoten am besten unterstützen, ist deshalb ein interessanter Aspekt bei der Entwicklung von Hochleistungsprotokollen. [DDK+90] und [LS91a] geben eine gute Übersicht über die Eignung der Protokollfunktionen für Hochleistungstransportprotokolle.

Bei der Bewertung der Protokollfunktionen muß berücksichtigt werden, daß der Empfänger eines Datenstroms einen höheren Berechnungsaufwand benötigt als der Sender und deshalb den Leistungsengpaß bildet (vgl. [NRS90]). Entscheidend dafür ist die Aufgabe, Nutzdaten und Steuerinformation einer empfangenen PDU voneinander zu separieren und den unterschiedlichen Protokollmechanismen die für sie relevanten Teile der Steuerinformation zugänglich zu machen. Netravali et al. [NRS90] schlagen deshalb vor, Nutzdaten und Protokollsteuerungsinformation in voneinander getrennten PDUs zu versenden und in jeder Steuerungs-PDU immer die gesamte Zustandsinformation zu übertragen. Der einfache und parallele Zugriff auf die Steuerinformation sollte durch ein einheitliches Format aller PDU-Köpfe unterstützt werden (vgl. [Che87]). Die Protokollsteuerinformation und die Nutzdaten sollten durch separate Prüfsummen gegen Übertragungsfehler gesichert werden, die am Paketende angeordnet sind, um einen sofortigen Vergleich mit der berechneten Prüfsumme durchführen zu können (vgl. [LS91a]). Um den Empfänger von der aufwendigen Verwaltung

von Zeitgebern zu entlasten, sollte der Transfer von Steuerinformation immer vom Sender angefordert werden.

Eine weitere Maßnahme zur Optimierung der Fehlerbehebung ist die Einführung von PDU-Blöcken, in denen pro Block immer nur ein Zähler verwendet wird. Damit kann man die Anzahl an Zeitgebern vermindern (vgl. [CLZ87, NRS90, Pro89]). Auch die Bestimmung geeigneter „Time Out"-Zeiten wird durch PDU-Blöcke vereinfacht, da die mittlere Übertragungsverzögerung eines Blocks weniger variiert als die Übertragungsverzögerung einer einzelnen PDU (vgl. [Zha86]). Bei der Meldung von Bitfehlern ist die Verwendung von expliziten Zurückweisungen (ARQ) mit vom Empfänger verwalteten Zeitgebern vorzuziehen, da der Empfänger die mittlere Übertragungsverzögerung anhand der ankommenden PDUs einfacher berechnen und die „Time Out"-Zeit seiner Zeitgeber einfacher optimieren kann (vgl. [CLZ87, Zha86]). Allerdings wird er durch die Zeitgeberverwaltung zusätzlich belastet, so daß das Verfahren gegenüber dem Austausch der gesamten Zustandsinformation von Nachteil ist.

Bei der Auswahl von Protokollfunktionen muß berücksichtigt werden, daß Pufferüberläufe und nicht Datenverfälschungen die häufigste Ursache für Übertragungswiederholungen sind (vgl. [CW89, Jac88]). Deshalb eignen sich selektive Übertragungswiederholungen eher als das Go-back-n-Verfahren [LS91a]. Bei einem Engpaß des Empfängerpuffers und daraus resultierenden häufigen PDU-Verlusten müßten nach dem Go-back-n-Verfahren aufgrund der hohen Bandbreite sehr viele PDUs wiederholt werden. Ohne eine Minderung des Datenflusses steigt dann aber die Netzbelastung und damit die Gefahr weiterer Datenverluste. Bei hoher Auslastung der Netzwerk- und Empfängerpuffer können mit dem Go-back-n-Verfahren nur 50 % der durch die selektive Übertragungswiederholung erreichbaren Übertragungsrate erreicht werden (vgl. [CW89]).

In einigen Protokollen (z.B. TCP) werden Fehlerbehandlungs- und Flußkontrollfunktionen miteinander verbunden. Die Bestätigung einer Anzahl von Dateneinheiten ist dabei zugleich die Erlaubnis, eine gleich große Anzahl von neuen Dateneinheiten zu senden (Schiebefensterprinzip, vgl. [Tan89]). Diese Verbindung ist jedoch in Hochleistungsprotokollen nicht sinnvoll, da die Protokollfunktionen der Flußkontrolle eine andere Strategie verfolgen als die Protokollfunktionen der Bestätigung (vgl. [NRS90]). Um eine gute Datenübertragungsrate sicherzustellen, sollen aus der Sicht der Flußkontrolle möglichst große Sendefenster für Dateneinheiten gewählt werden. Wenn jedoch die Größe des Sendefensters die Pufferkapazität des Empfängers überschreitet, steigt die Wahrscheinlichkeit von Datenverlusten. Deshalb sind aus Sicht der Fehlerbehandlungsprotokollfunktionen kleinere Sendefenster wünschenswert. Bei Einsatz des Go-back-n-Verfahrens steigt zudem bei großen Fenstern die mittlere Anzahl der zu wiederholenden Dateneinheiten, und die effektive Datenübertragungsrate sinkt (vgl. [LS91a]).

Bei der Realisierung der Flußkontrolle sind auf temporären Sendebeschränkungen basierende Protokollfunktionen gegenüber Kreditvergabeschemata vorzuziehen (vgl. [Mor88]), da sie eine weitgehende Entkoppelung des Datenflusses vom Protokollsteuerungsfluß erlauben. Explizite Mitteilungen sind nur beim Anpassen der Senderate notwendig. Eine gute Datenübertragungsrate erreichen auch Systeme, in denen beide Schemata verwendet werden (vgl. [LS91a]). Außerdem kann die Belastung des unterliegenden Netzwerks (Access Control) nur durch Methoden der temporären Sendebeschränkung beeinflußt werden.

In Abschnitt 2.2.3 wurden implizite und auf Bestätigungen basierende Verfahren zum Verbindungsaufbau und -abbau vorgestellt. Implizite Verfahren (oder verbindungslose Übertragung) eignen sich hervorragend für Anwendungen, in denen kleine Datenmengen mit ei-

ner geringen Übertragungsverzögerung gesendet werden sollen (z.B. verteilte Verarbeitung). Insbesondere in Weitverkehrsnetzen würde bei bestätigtem Verbindungsaufbau und -abbau durch die lange Laufzeit der Bestätigungen eine schlechte effektive Übertragungsrate erreicht[1]. Bei Anwendungen, die über eine längere Zeit bestehen (z.B. Videoübertragungen), ist der Einfluß eines bestätigten Verbindungsmanagements geringer, das aufgrund seiner geringeren Störanfälligkeit in diesem Fall vorzuziehen ist. Hochleistungsprotokolle sollten also sowohl den impliziten als auch den bestätigten Verbindungsaufbau und -abbau unterstützen (vgl. [DDK+90, LS91a]).

2.3.4 Optimierung der Protokollgraphen

Einige Protokollfunktionen sind in verschiedenen Schichten des ISO/OSI-Basisreferenzmodells [ISO94] vorhanden. Zum Beispiel werden Maßnahmen zur Behebung von Bitfehlern in der Sicherungsschicht, der Vermittlungsschicht und der Transportschicht eingesetzt. Methoden zur Flußkontrolle können in diesen Schichten und zusätzlich in der Anwendungskommunikationsschicht auftreten (vgl. [Fel93]). Die strenge Abgrenzung der Protokollinstanzen auf den Schichten führt jedoch dazu, daß für die Realisierung der Protokollfunktionen mehrere gleichartige Protokollmechanismen notwendig werden und deshalb effektive Rechenzeit überflüssigerweise verbraucht wird. Kommunikationssysteme sollten also so ausgelegt werden, daß jede Protokollfunktion nur einmal implementiert wird. Fehlerbehebung sollte zum Beispiel nur auf einer Schicht des Endsystems und nicht an den Vermittlungsknoten durchgeführt werden (vgl. [Fel93, SRC84]). Damit werden die Puffer zum Ordnen vertauschter Daten und zur Desegmentierung nur an einer Stelle implementiert, und der rechenzeitintensive Zugriff auf diese Daten erfolgt nur einmal (vgl. [CT90, OP91, PGP92]). Das Multiplexing von Verbindungen sollte ebenfalls nur an einer Stelle im System erfolgen, da dadurch ebenfalls Rechenzeit — wenn auch auf Kosten zusätzlichen Speicherplatzes — gespart wird (vgl. [Fel90, McA90, Ten89]). An dieser Stelle kann auch die einzige Instanz der Ende-zu-Ende-Flußkontrolle angeordnet werden (vgl. [Fel90]).

Da diese Vereinfachungen jedoch nicht durch die vom ISO/OSI-Basisreferenzmodell geforderten Architekturen realisiert werden können, wird in diversen Ansätzen untersucht, wie man die Protokollschichtung durch flexiblere Strukturen ersetzen kann.

Green [Gre91] empfiehlt die prinzipielle Beibehaltung des Basisreferenzmodells, wobei jedoch die Vermittlungsschicht und ein Teil der Sicherungsschicht (Logical Link Control-Teilschicht, LLC) in die Transportschicht eingegliedert werden. Jede Protokollfunktion, die er als „atomare Funktion“ (atomic function) bezeichnet, soll nur in jeweils einer Schicht realisiert werden. Zum Beispiel sollen Flußkontrolle nur in der erweiterten Transportschicht und Segmentierung nur in der Kommunikationssteuerungsschicht erfolgen. Die Maßnahmen der Fehlerbehebung werden in Abweichung von der im ISO/OSI-Basisreferenzmodell gegebenen Zuordnung vollständig der Kommunikationssteuerungsschicht zugeordnet.

Clark und Tennenhouse [CT90] schlagen dagegen in ihrem Integrationsansatz (Integrated Layer Processing, ILP) die Zusammenlegung der anwendungsorientierten Schichten vor. Nach ihrer Meinung bilden die Datenkonversionen zwischen lokaler Syntax und Transfersyntax einen wesentlichen Leistungsengpaß. Die Anpassung empfangener Nutzdaten an das

[1] [LS91a] nennt für die Übertragung von einem 1 MByte großen Datenstrom über eine Entfernung von 4885 km auf einem Rechnernetz von einer Bandbreite von 1 GBit/s eine effektive Datenübertragungsrate von nur 60 MBit/s.

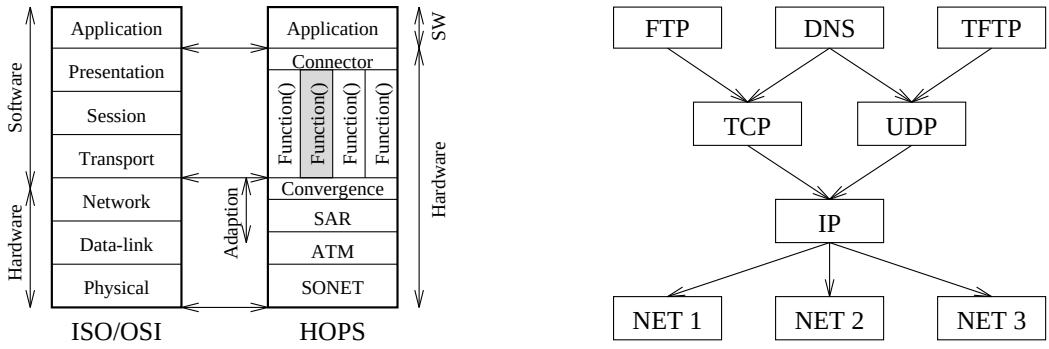

Abbildung 2.4: HOPS-Struktur (entnommen aus [Haa91] mit Vereinfachungen) und TCP/IP-Protokollgraph (entnommen aus [OP92])

lokale Format der Empfängeranwendung (z.B. Konversion der Transfersyntax in die lokale Syntax, Überprüfung der Prüfsumme, Entschlüsselung) beansprucht ebenfalls viel Rechenzeit. Diese Konversionsfunktionen sind im Basisreferenzmodell verschiedenen Schichten zugeordnet und können deshalb nicht zueinander parallel ausgeführt werden. Günstig wäre also, sie vollständig der Anwendungsschicht zuzuordnen. Um ferner Verzögerungen der Datenkonversion durch vertauschte oder verlorengegangene Dateneinheiten auszuschließen, sollte auch die Fehlerbehandlung in der Anwendungsschicht erfolgen.

Einen ähnlichen Ansatz propagiert Haas, der anstatt der Separierung in Transportschicht, Kommunikationssteuerungsschicht und Darstellungsschicht eine horizontale Struktur (Horizontally Oriented Protocol Structure, HOPS) empfiehlt [Haa91] (siehe linke Seite in Abbildung 2.4). Jede Komponente dieser Struktur realisiert genau eine Protokollfunktion. Als Protokollfunktionen schlägt er Fehlerbehandlung, Sendewiederholung, Verbindungsbehandlung, Paketsequenzen, Flußkontrolle, Adressierung, Darstellung, Verbindungssteuerung und Staukontrolle vor, die über separate Felder in der Protokollsteuerinformation (PCI) der PDUs mit komplementären Funktionen in anderen Stationen kommunizieren. Diese Architektur erlaubt zum einen, Protokolle einfach an die entsprechende Dienstgüte anzupassen, indem nur ein bestimmter Teil der Protokollfunktionen verwendet wird (siehe Abschnitt 2.3.5). Zum anderen können die Protokollfunktionen einfach parallelisiert oder durch Hardware-Bausteine realisiert werden (siehe Abschnitt 2.3.6). Auch wird sichergestellt, daß diese Protokollfunktionen nur an einer einzigen Stelle im Kommunikationssystem realisiert werden.

O'Malley und Peterson [OP91, OP92] empfehlen hingegen eine Rechnerarchitektur mit vielen Schichten. Dabei soll für jede Protokollfunktion eine eigene Schicht entworfen werden. Die Teile eines Protokolls, die eine Protokollfunktion erbringen, werden „Mikroprotokolle" genannt (siehe rechte Seite in Abbildung 2.4). Im Gegensatz zum ISO/OSI-Basisreferenzmodell ist der Protokollgraph kein starres Gebilde aus Schichten, sondern kann in Abhängigkeit der Dienstgüte dynamisch konfiguriert werden (siehe 2.3.5). Dazu werden „virtuelle Protokolle" eingeführt, deren Aufgabe eine konfigurationsabhängige Weiterleitung von Nutzdaten an bestimmte Mikroprotokollinstanzen ist. Anstatt sequentieller Protokollgraphen werden verzweigte Architekturen realisiert, in denen der Datenfluß an die geforderte Dienstgüte angepaßt werden kann. Das Ziel, jede Protokollfunktion nur einmal zu implementieren, kann dadurch

erreicht werden, daß im Protokollgraph für jede Protokollfunktion nur ein Mikroprotokoll vorhanden ist.

2.3.5 Protokollkonfigurierung

In Abschnitt 2.3.1 wurde gezeigt, daß sich die typischen Hochleistungsrechnernetzanwendungen in ihren quantitativen und funktionalen Dienstgüteanforderungen voneinander zum Teil sehr unterscheiden. Ein universelles Hochleistungskommunikationssystem muß also eine große Anzahl an Protokollfunktionen realisieren, um die unterschiedlichen Anforderungen zu unterstützen, obwohl jede einzelne Anwendung meist nur einen Teil dieser Funktionalität nutzt. Zum Beispiel reicht die Bitfehlerrate von Lichtwellenleitern für Bewegtbild- und Sprachübertragungen aus, so daß keine weitere Fehlerbehandlungsmaßnahmen notwendig sind. Um den überflüssigen Einsatz von Protokollfunktionen zu verhindern, werden Ansätze verfolgt, Protokolle in Abhängigkeit der vom Anwender geforderten Dienstgüte und der Qualität des verwendeten Rechnernetzes zu konfigurieren. Feldmeier spricht in diesem Zusammenhang von quasi-statischer Protokollkonfigurierung (vgl. [Fel93]).

Eine Lösung besteht darin, dem Dienstnehmer anstatt nur eines einzigen Protokolls ein ganzes Protokollspektrum zur Verfügung zu stellen. Die Auswahl eines geeigneten Protokolls erfolgt beim Verbindungsaufbau abhängig von der geforderten Transportdienstgüte und den Netzwerkeigenschaften. Ein Beispiel dafür sind die fünf Klassen des ISO/OSI-Transportprotokolls [ISO86a], die sich in den Fehlerbehandlungs- und Multiplexingprotokollfunktionen unterscheiden. Die Auswahl einer Klasse erfolgt dabei in Abhängigkeit von der Restfehlerrate und der Qualität des Vermittlungsdienstes. Sterbenz und Parulkar [SP90] schlagen mit ihrer Protokollarchitektur „Axon“ eine ähnliche Struktur für Protokolle vor, die jedoch die Anwendungen weitergehend berücksichtigt. Für unterschiedliche Anwendungsprofile stehen eigene Hochleistungsprotokolle zur Verfügung, die nur die jeweils notwendigen Protokollfunktionen enthalten. Das MultiStream-Protokoll (MSP) von La Porta und Schwartz [LS93] bietet in jeder Transportverbindung sieben Transportkanäle an, die verschiedenartige Dienstgüten erbringen. Die Transportkanäle verwenden dazu unterschiedliche Protokollfunktionen für Block-/Einzelpaketsendung, Bestätigung, Sendewiederholung und Reihenfolgesicherung.

Quasi-statische Protokollkonfiguration kann jedoch auch in Protokollen eingesetzt werden, die nicht auf einzelne Anwendungen zugeschnitten sind (vgl. [WM87]). Beispiele sind die in Abschnitt 2.3.4 beschriebenen Ansätze von Haas [Haa91] bzw. O'Malley und Peterson [OP91], in denen die einzelnen Protokollfunktionen bzw. Mikroprotokolle in Abhängigkeit von der geforderten Dienstgüte ausgewählt werden können. Tschudin [Tsc91] stellt eine Methode vor, Protokollgraphen dynamisch zu konfigurieren. Er führt „Protocol Stack Environments“ ein, die aus einer Anzahl von Protokollinstanztypen bestehen, die explizit instantiiert und gelöscht werden. Die Kopplung zwischen den einzelnen Protokollinstanzen kann dabei je nach Anwendung verändert werden. In einem weitergehenden Ansatz schlagen Zitterbart et al. [ZST93] ein Modell zur Verwaltung und Konfigurierung von Protokollfunktionen vor, das sie „Function-Based Communication Subsystem“ (F-CSS) nennen. Es sieht einen Pool aus Protokollressourcen vor, der aus Protokollfunktionen, der „Protocol Function Information Base“ (PFIB) und einem Protokollmanager besteht. PFIBs enthalten Protokollsteuerdaten und Nutzdateneinheiten. In Abhängigkeit von der geforderten Dienstgüte generiert der Protokollmanager aus den Protokollfunktionen und PFIBs eine lauffähige Protokollinstanz, die nach Beendigung der Datenkommunikation wieder aufgelöst wird. Einen ähnlichen Ansatz verfolgt

das Projekt Da CaPo (Dynamic Configuration of Protocols) von Plagemann et al. [PPVW93].

2.3.6 Parallelisierung von Protokollfunktionen

Ein wichtiges Konzept zur Parallelisierung von Protokollfunktionen ist Pipelining von Datenströmen (vgl. [GKWW89, Zit91]. Nutzdaten werden bei ihrem Durchlauf durch den Protokollgraphen nacheinander zu den einzelnen Protokollfunktionen gereicht, die auf eigenen Prozessoren ablaufen. Die Bearbeitungsgeschwindigkeit der gesamten Pipeline wird durch die langsamste Protokollfunktion bestimmt (vgl. [Zit89]). Zitterbart [Zit91] unterscheidet zwei Pipelines für die Sendung bzw. den Empfang von Dateneinheiten, die voneinander relativ unabhängig ablaufen, allerdings an einigen Stellen miteinander interagieren. Braun und Zitterbart zeigen in [BZ93], wie man Protokollfunktionen entwirft, die nur über sehr wenige Interaktionspunkte verfügen, so daß sie gut parallelisiert werden können. Diese Struktur wird im Protokoll PATROCLOS [Bra93] realisiert.

2.4 Einige Beispiele von Hochleistungsprotokollen

Nachdem wir in Abschnitt 2.3 einen Überblick über die Eigenschaften von Hochleistungstransportprotokollen gegeben haben, möchten wir an dieser Stelle einige Protokollentwicklungen aufzeigen. Zunächst stellen wir kurz die wichtigsten Übertragungsverfahren vor, um einen Eindruck von der Beschaffenheit der untergeordneten Netzwerke zu geben, auf denen die Transportprotokolle aufsetzen. Anschließend werden wir aktuelle Transportprotokollentwicklungen erläutern. Inzwischen ist die Anzahl der Ansätze so groß geworden, daß eine vollständige Aufzählung den Rahmen dieser Arbeit sprengen würde. Wir beschränken uns deshalb auf die bekanntesten Protokolle und einige weitere Entwicklungen, die nach unserer Meinung interessante Neuerungen in Bezug auf die Protokollarchitektur einführen. Wir werden die Protokolle NETBLT, VMTP, XTP, SNR, PATROCLOS und MultiStream skizzieren (siehe auch Abbildung 2.5).

2.4.1 Übertragungsverfahren

Wie bei den Transportprotokollen (siehe Abschnitt 2.3.2) wurde auch in dieser Protokollklasse versucht, konventionelle Standardprotokolle an die Bedürfnisse von Hochleistungsrechnernetzen anzupassen. Ein Beispiel dafür ist Fast-Ethernet, das auf Lichtwellenleitern (oder verdrillten Kupferadern) aufsetzt und eine Übertragungsrate von 100 MBit/s anbietet. Der Medienzugriff verläuft wie im konventionellen Ethernet nach dem CSMA/CD-Standard [IEE96].

Eine Weiterentwicklung des Token-Rings [IEE95] ist das Übertragungsverfahren FDDI (Fiber Distributed Data Interface), das paketvermittelten Datentransfer mit einer Übertragungsrate von 100 MBit/s realisiert [ANS92]. Es verwendet zwei aus Lichtwellenleitern bestehende Ringe, wovon einer nur zur Ausfallsicherung dient. Der Medienzugriff erfolgt wie beim Token-Ring über eine kreisende Berechtigungsmarke. Eine Station darf nur Daten senden, wenn sie Zugriff auf eine nicht belegte Berechtigungsmarke hat. Im Gegensatz zum Token-Ring gibt eine Station die Marke allerdings sofort nach Beendigung der Sendung weiter, so

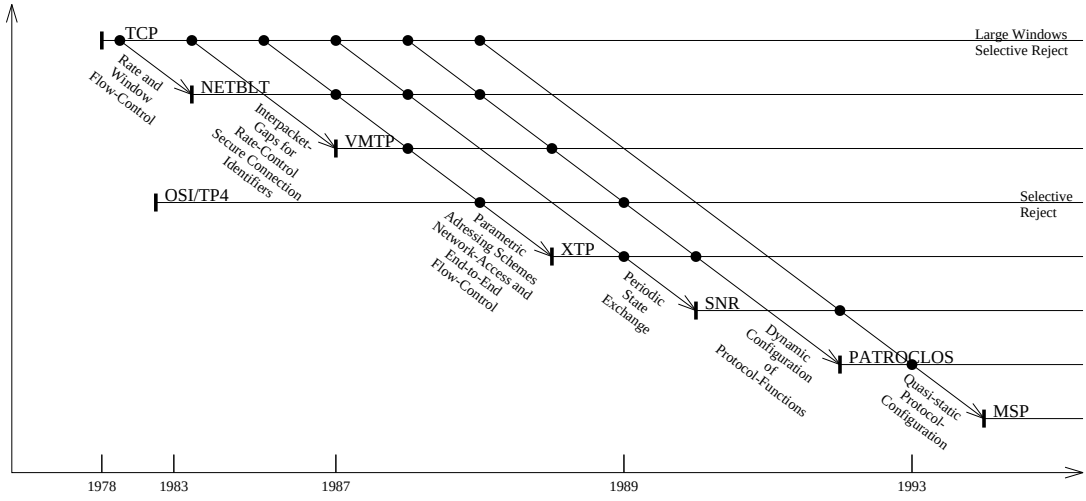

Abbildung 2.5: Evolution der Hochleistungstransportprotokolle (erweitert entsprechend der Vorlagen aus [DDK+90, BR92])

daß unter Umständen mehrere Pakete gleichzeitig auf dem Netz sein können. Ein Paket wird von seinem Sender vom Ring genommen.

Eine Neuentwicklung für Hochleistungsrechnernetze sind das Übertragungsverfahren ATM und die darauf basierenden Netzwerk-Standards B-ISDN und DQDB, denen allgemein eine weite Verbreitung vorausgesagt wird. ATM (Asynchronous Transfer Mode) ist ein Übertragungsverfahren, das seinen Benutzern eine schnelle verbindungsorientierte Vermittlung von Zellen ohne Fehlerbehandlung und Flußkontrolle mit einer Übertragungsrate von bis zu 662 MBit/s (später 2,4 GBit/s) bietet [dP94, Zit95]. Eine Zelle ist ein kleines Datenpaket mit einer festen Größe von 53 Byte, wovon höchstens 48 Byte für Nutzdaten verwendet werden können. Benutzer können untereinander eine große Anzahl unterschiedlicher virtueller Kanäle bzw. Pfade unterhalten.

Auf der Basis von ATM werden Standards für lokale Netze (vgl. [Kun92, LS92]), regionale Netze und Weitverkehrsnetze entwickelt. Als Standard für regionale Netze wurde DQDB (Distributed Queue Dual Bus) entworfen. DQDB ist im Rahmen des IEEE-Standards 802 festgelegt [IEE94] und besitzt einen gleichartigen Netzwerkzugriff wie die bekannten lokalen Netze (z.B. Ethernet, Token-Bus, Token-Ring). DQDB sieht verbindungsorientierten und verbindungslosen Datentransfer vor. Die Netzwerktopologie besteht aus einer Reihe von Stationen, die über zwei unidirektionale Datenbusse mit unterschiedlichen Datenflußrichtungen gekoppelt sind. Die Station am Anfang eines Busses sendet Rahmen, die einzelne ATM-Zellen aufnehmen können. Die Rahmen werden von den Stationen beschrieben und gelesen und am Ende des Kanals vernichtet. Der Zugriff auf die Rahmen erfolgt über ein Protokoll, in dem eine Station i, die eine Zelle über Kanal A senden möchte, zuvor eine Reservierungsanforderung über Kanal B sendet. Die Empfänger dieser Anforderung, die im Kanal A vor i angeordnet sind, lassen dann einen Rahmen auf Kanal A unbelegt zu Station i durch, die ihre Zelle dann darauf senden kann. Um fairen Zugriff zu garantieren, bilden die Stationen eine virtuelle Warteschlange (vgl. [Zit95]).

Das auf (Schmalband-)ISDN basierende Weitverkehrsnetz B-ISDN (Broadband-Integrated Digital Network) [ITU92] setzt ebenfalls auf ATM auf. Außer dem ATM-Zugriff bietet B-ISDN weitergehende Dienste an, die die von den Anwendungen geforderten über ATM hinausgehende Dienstgüten erbringen. B-ISDN sieht vier Dienstklassen vor, die Dienste nach drei Hauptparametern unterscheiden: Notwendigkeit einer Realzeitbeziehung zwischen Sender und Empfänger, konstante oder variable Bitrate, verbindungsloser oder verbindungsorientierter Verbindungsmodus (vgl. [dP94]). Weitere Protokollfunktionen erlauben die Nutzung von Segmentierung, Sequenzzahlen, Multiplexing und Fehlerbehebung. Der Anschluß von Stationen an B-ISDN kann durch unterschiedliche Topologien (Stern, Bus, Ring) erfolgen (vgl. [Zit95]). DQDB kann aufgrund seiner Kompatibilität ebenfalls als Vorschalt-Netzwerk von B-ISDN betrieben werden.

2.4.2 NETBLT

Das Transportprotokoll NETBLT (Network Block Transfer Protocol) [CLZ87] wurde ursprünglich für die Satellitenkommunikation und andere Übertragungswege mit großer Laufzeitverzögerung entwickelt (vgl. [BR92]). Wie Hochleistungsnetzwerke zeichnet sich die Satellitenkommunikation durch eine hohe Pfadkapazität aus, so daß in beiden Bereichen gleichartige Protokollfunktionen eingesetzt werden. NETBLT eignet sich damit auch gut für Hochleistungsrechnernetze.

Als erstes Protokoll verwendet es Flußkontrolle auf der Basis temporärer Sendebeschränkungen mit datenratenbasierter Sendesteuerung (siehe Abschnitt 2.2.2). Beim Verbindungsaufbau handeln die Protokollinstanzen die Anzahl an Bytes aus, die in einer ununterbrochenen Folge (Burst) gesendet werden dürfen (Burst-Size), und die Anzahl an Bursts, die in einer Zeiteinheit gesendet werden dürfen (Burst-Rate). Zusätzlich bietet NETBLT ein auf Sendekredit basierendes Flußkontrollverfahren zum Schutz des Empfängers vor Pufferüberläufen an.

Zur Behebung von Bitfehlern verwendet NETBLT selektive Datenwiederholung. Aufgrund der hohen Pfadkapazität existiert in der Regel eine große Anzahl an Dateneinheiten, die zwar gesendet wurden, aber noch nicht bestätigt sind. Um nicht für jede dieser Dateneinheiten einen eigenen Zeitgeber für die Wiederholungssteuerung zu verwalten, ermöglicht NETBLT auch die Zusammenfassung von Dateneinheiten zu Blöcken.

Insgesamt gesehen, verwendet NETBLT effiziente Protokollfunktionen für die Flußkontrolle und Fehlerbehebung (siehe Abschnitt 2.3.3). Es ist allerdings aufgrund seiner auf Bestätigungen basierenden Verbindungsaufbau- und Verbindungsabbaumechanismen weniger für Anwendungen geeignet, in denen häufig kurze Nachrichten übermittelt werden müssen (z.B. verteilte Berechnungen).

2.4.3 VMTP

Ein wesentliches Merkmal von VMTP (Versatile Message Transaction Protocol) [Che88, CW89] ist die Unterstützung schneller transaktionsorientierter Anwendungen (z.B. Remote Procedure Calls, RPCs). Im Gegensatz zu NETBLT unterstützt es damit schnellen impliziten Verbindungsaufbau und durch Zeitgeber gesteuerten Verbindungsabbau.

Um auch den Transfer großer kontinuierlicher Datenströme zu ermöglichen, verwendet VMTP wie NETBLT Fehlerbehebung mit selektiver Datenwiederholung und Flußkontrolle

durch temporäre Sendebeschränkungen (vgl. [Che86]). Die Flußkontrolle unterscheidet sich von der in NETBLT darin, daß die Senderatenanpassung durch Sendepausen zwischen zwei gesendeten PDUs erfolgt, die durch einen in Hardware realisierten Zeitgeber mit sehr feiner Granularität gesteuert wird (Interpacket Delay). Anhand der Häufigkeit des Auftretens von Sendewiederholungen, aus der die Netzauslastung geschätzt wird, kann der Sender die Sendepausenlängen modifizieren.

Bei der Entwicklung des Paketformats von VMTP wurde darauf Wert gelegt, daß Protokollfunktionen durch Hardwarebausteine realisiert werden können (vgl. [BR92]). Eine Netzwerkadapterkarte des VMTP-Protokolls wurde bereits entwickelt (vgl. [KC88]). Ein weiteres Merkmal von VMTP ist seine Adressverwaltung, die, basierend auf ortsunabhängigen Adressen, leistungsfähige Mechanismen zur Sicherstellung der Adreß-Authentizität anbietet.

2.4.4 XTP

XTP ((e)Xpress Transfer Protocol) [Pro89, Che91] wurde entwickelt, um Anwendungen mit variabler Dienstgüte auf Hochleistungsrechnernetzen mit unterschiedlicher Leistungsfähigkeit zu unterstützen. Wesentliche Protokollfunktionen wie Verbindungsaufbau, -abbau, Fehlerbehandlung und Flußkontrolle können von den Dienstnehmern frei gewählt werden (vgl. [SW90, Wea94]). Die Verbindungsverwaltung läßt sowohl virtuelle Verbindungen als auch bestätigte oder unbestätigte Datagramme zu. Außerdem werden Mehrbenutzerverbindungen unterstützt.

Zur Meldung von Bitfehlern können sowohl Funktionen der positiven Bestätigung als auch der schnellen sequentiellen Zurückweisung (siehe Abschnitt 2.2.2) eingesetzt werden. Die Fehlerbehebung erfolgt durch selektive Wiederholung oder Rücksetzung. Außerdem kann die Fehlerbehandlung ganz abgeschaltet werden. Durch die Konfigurierbarkeit werden unterschiedliche Anforderung wie die Übertragung langer Dateien, häufiger kurzer Nachrichten oder eines kontinuierlichen Bewegtbildstroms gut unterstützt.

Auch die Flußkontrolle kann an die Bedürfnisse der Protokollanwendungen angepaßt werden. Neben einer auf temporärer Sendebeschränkung basierenden Flußkontrolle kann zusätzlich ein Kreditmechanismus eingesetzt werden, um Pufferüberläufe beim Empfänger zu verhindern. Die Flußkontrolle kann aber auch ganz abgeschaltet werden. XTP führt als erstes Protokoll eine eigenständige Flußkontrolle für das untergeordnete Medium ein (Access Control), die auf temporären Sendebeschränkungen basiert.

Hauptziel der Entwicklung von XTP ist die Realisierung einer hohen Datenübertragungsrate. Die Trennung der ISO/OSI-Transportschicht und der ISO/OSI-Vermittlungsschicht wurde aufgehoben, so daß die in beiden Schichten implementierten Protokollfunktionen nur einmal realisiert werden müssen (siehe auch Abschnitt 2.3.4). Um die Hardware-Implementierung wichtiger Protokollfunktionen wie Fehlererkennung und Paketkopfgenerierung auf VLSI-Chips zu ermöglichen, besitzt XTP Paketformate mit Feldern von fester Länge (vgl. [BR92]). Es kann damit auch gut auf Mehrprozessor-Architekturen realisiert werden (vgl. [BSZ92]).

Aus heutiger Sicht scheint sich XTP als Transportprotokoll für Hochleistungsnetzwerke durchzusetzen. Da es zudem alle Eigenschaften moderner Transportprotokolle besitzt, wurde die aktuelle Version 4.0 [XTP95] von uns als Anwendungsbeispiel für den kompositionalen Spezifikationsstil ausgewählt (siehe Kapitel 6).

2.4.5 SNR

Eine strenge Trennung von Daten- und Protokollsteuerfluß ist im Protokoll SNR [NRS90] vorgesehen. Der Sender und der Empfänger senden einander in kurzen periodischen Abständen PDUs zu, in denen alle für den Partner relevanten Daten über den lokalen Instanzzustand enthalten sind. Das Protokoll sieht vor, die Protokollfunktionen durch eigene Prozesse zu realisieren, die aufgrund des festen PDU-Formats einfach mit der für sie relevanten Protokollsteuerinformation (PCI) versorgt werden können. Durch separate Bezeichner kann man Steuer-PDUs einfach von Daten-PDUs unterscheiden. Außerdem hat jede Protokollinstanz eine konsistente Sicht ihrer Partnerinstanzen.

Das Protokoll unterstützt typische Hochleistungsprotokollfunktionen wie Fehlerbehebung durch selektive Datenwiederholung. Es unterstützt drei operationale Modi: weder Fehlerbehebung noch Flußkontrolle, nur Fehlerbehebung, sowohl Fehlerbehebung als auch Flußkontrolle.

2.4.6 PATROCLOS

Die parallele Bearbeitung von Protokollfunktionen durch eigene Prozesse ist wesentliches Entwurfskriterium von PATROCLOS (Parallel Transport System for Cell Based High Speed Networks) [Bra93]. Die Protokollinstanzen werden in modulare Protokollautomaten zerlegt (vgl. [BZ93]). Zwei Schnittstellenautomaten erbringen den Datentransfer an den Schnittstellen zum Transportdienstnehmer und zum Transfermedium. Die Protokollfunktionen werden durch Protokollautomaten realisiert. PATROCLOS sieht Protokollautomaten für die Verbindungsverwaltung, für den Nutzdatenaustausch, für Bestätigung und Fehlerbehandlung, für die Ende-zu-Ende-Flußkontrolle und für die Lastkontrolle des Mediums vor. Die Aufteilung der Protokollmechanismen auf die Protokollautomaten wurde dabei so ausgelegt, daß die Interaktion zwischen den Automaten minimal ist (siehe auch Abschnitt 2.3.6). Im Gegensatz zu SNR existiert für jede Protokollautomatenklasse ein eigener PDU-Typ.

PATROCLOS kann stärker an die angeforderte Dienstgüte angepaßt werden als andere Protokolle. Beim Verbindungsaufbau werden implizite, einfach bestätigte und doppelt bestätigte Verfahren unterstützt. Zum Verbindungsabbau stehen Protokollfunktionen zur Verfügung, die auf Zeitgebersteuerung, auf einfacher Bestätigung oder auf doppelter Bestätigung basieren. Die Reassemblierung segmentierter Daten kann abgeschaltet werden. Die Übertragungswiederholung kann per selektiver Wiederholung oder per Rücksetzung erfolgen. Außerdem kann sie ebenfalls abgeschaltet werden. Für die Flußkontrolle sind „Stop-and-Wait“-Verfahren und Sendekredite vorgesehen. Zusätzlich dazu kann ein auf temporärer Sendebeschränkung basierender Flußkontrollmechanismus hinzugefügt werden. Sie kann aber auch ganz abgeschaltet werden. Die Kontrolle der Netzbelastung kann ebenfalls ausgeschaltet oder nach einem von zwei Systemlasterkennungsverfahren (2P-Algorithmus, AACC-Algorithmus) betrieben werden.

2.4.7 MSP

Quasi-statische Konfigurierbarkeit (siehe Abschnitt 2.3.5) ist eine wesentliche Eigenschaft des MSP (MultiStream-Protokoll) [LS93]. Bei einem Verbindungsaufbau werden dem Benutzer sieben parallele Datentransportkanäle („Streams“) zur Verfügung gestellt, die unterschiedliche Dienstgüten anbieten. Als Protokollfunktionen zur Fehlermeldung werden je nach Kanal kumulative Bestätigungen mit zeitgebergesteuerter Wiederholung, selektive Zurückweisung

Daten-transport-kanal	Protokollfunktionen gegen Blöcke	Bit-fehler	Ver-luste	Ver-tausch.	Protokollfunktionen Fehler-erkennung	Fehler-behebung
1		*	*	*	KB	R
2				*	nichts	nichts
3		(*)			alles	alles
4		*		*	KB/SR	R/SW
5	*			*	nichts	nichts
6	*	*	*		KB/SR	R/SW
7	*	*	*	*	KB/SB	R/SW

Legende:
* – Protokollfunktion ist vorhanden
(*) – Protokollfunktion wählbar
nichts – keine Protokollfunktion verfügbar
alles – freie Auswahl aller Protokollfunktionen
KB – Kumulative Bestätigungen
SR – Selektive Zurückweisungen
R – Rücksetzung
SW – Selektive Wiederholung

Tabelle 2.1: Protokollfunktionen und Fehlerbehebungsmaßnahmen in MSP

oder keine Fehlermeldung eingesetzt. Die Fehlerbehebung erfolgt durch selektive Wiederholung, Rücksetzung oder überhaupt nicht. Verbindungen können entweder implizit oder durch doppelte Bestätigungen aufgebaut werden. Die Funktionalität der einzelnen Kanäle und die verwendeten Protokollmechanismen sind in Tabelle 2.1 aufgeführt.

2.5 Einige Gedanken zur Modellierung von Hochleistungsprotokollen

Wir haben dargestellt, daß die Schaffung rechenzeiteffizienter Protokollinstanzen das Kardinalproblem bei der Entwicklung von Hochleistungsprotokollen ist. Zur Lösung dieses Problems existieren viele unterschiedliche Ansätze (siehe Abschnitt 2.3) und auch schon eine größere Anzahl an Realisierungen (siehe Abschnitt 2.4). Bis jetzt ist noch nicht klar, welcher Ansatz und welche Realisierung sich als Standard für Hochleistungstransportprotokolle durchsetzen wird. Allerdings gibt es Anzeichen dafür, daß XTP (siehe Abschnitt 2.4.4) Grundlage für einen Standard werden könnte.

Bei der Entwicklung bestand bisher nur eine geringe Tendenz, die Ansätze formal zu modellieren oder sie gar auf ihre funktionale Korrektheit hin zu überprüfen (als Beispiele hierzu vgl. [CA90, Hei92, LS94, LT94, Sto93]). Ein Grund hierfür ist neben der inhärenten Komplexität der Modellierung und Überprüfung nebenläufiger Systeme die Unsicherheit der Entwickler bezüglich der Stabilität ihrer Entwürfe. Häufig kann erst bei der praktischen Überprüfung festgestellt werden, ob ein Ansatz zufriedenstellend ist oder ob er verändert werden muß. Eine Veränderung führt jedoch häufig dazu, daß Modelle und Protokollüberprüfungen überarbeitet werden müssen. Die Notwendigkeit, Protokolle formal zu modellieren, um Systeme hoher Qualität zu erlangen, ist dennoch unumstritten. Gerard Holzmann berichtet anläßlich des NewCoRe-Projekts, daß 55 % der originalen Entwurfsspezifikationen logisch inkonsistent waren [Hol94].

Die oben genannten Ansätze haben trotz ihrer großen Unterschiede eine gemeinsame Eigenschaft: Sie verwenden nur eine ziemlich kleine Anzahl an Protokollfunktionen, die ebenfalls nur durch relativ wenige Protokollmechanismen realisiert werden (siehe Abschnitt 2.2.2 und 2.2.3). Diese Protokollmechanismen werden auf sehr unterschiedliche Weise miteinander zu Protokollimplementierungen kombiniert. Ihre Kombination spielt dabei nur insofern eine Rolle, als sich die Protokollmechanismen bei der Interaktion nicht gegenseitig behindern.

Unter Berücksichtigung dieser Eigenschaften kann die formale Modellierung vereinfacht werden, indem man Protokollmechanismen als Modellbausteine spezifiziert. Die Kombination der Protokollmechanismen zu einem Protokoll entspricht dann der Komposition der Modellbausteine zu einer Spezifikation des Protokollsystems. Darüber hinaus wird die Untersuchung, ob ein Protokoll seinen Dienst realisiert, zerlegt. Für einzelne Diensteigenschaften (siehe Abschnitt 2.2.2) wird nachgewiesen, daß sie jeweils durch eine Kombination von Protokollmechanismen realisiert werden. Außerdem wird überprüft, ob die Kopplung der Protokollmechanismen störungsfrei ist. Diese Anforderungen an eine Modellierungstechnik werden genau durch den in Kapitel 4 vorgestellten kompositionalen Spezifikationsstil und das zugehörige Konzept der strukturierten Verifikation erfüllt.

Darüber hinaus kann man eine Bibliothek von Modellbausteinen zusammenstellen, die alle bekannten Protokollmechanismen und Diensteigenschaften umfaßt. Protokollentwickler können dann aus diesen Bausteinen die von ihnen gewünschten Protokollspezifikationen komponieren. Wenn die Bibliothek außerdem bereits bewiesene Theoreme enthält, daß eine bestimmte Komposition der Protokollmechanismenbausteine einen Diensteigenschaftenbaustein realisiert, beschränkt sich die Arbeit des Protokollentwicklers auf die Überprüfung, ob die Bausteine miteinander korrekt interagieren. Eine solche Bibliothek für das hier vorgestellte Gebiet der Hochleistungsprotokolle stellt das in Kapitel 5 vorgestellte Framework dar.

Kapitel 3

Formale und funktionale Entwurfsmethoden

Wie bereits in Kapitel 1 ausgeführt wurde, reicht die klassische „Ad-hoc“-Softwareentwicklung ohne methodische Unterstützung für heutige Software nicht mehr aus. Das gilt insbesondere für Systeme aus miteinander und mit ihrer Umgebung kommunizierenden Instanzen wie die hier betrachteten Hochleistungstransportprotokolle, die aufgrund ihrer Nebenläufigkeit und der daraus resultierenden Vielzahl an möglichen Systemabläufen meist sehr komplex sind (vgl. [AO91a]). Die Systementwicklung muß also durch Methoden unterstützt werden, die ein systematisches Vorgehen ermöglichen und damit zu einer besseren Bewältigung der Komplexität beitragen. Jeannette Wing sieht in formalen Methoden ein geeignetes Mittel für den systematischen Entwurf von Systemen:

> „Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc, manner.“ (Zitat aus [Win90])

Formale Methoden unterstützen sowohl die Beschreibung als auch die Analyse von Systemen. Formale Spezifikationstechniken verwenden eine eindeutige Syntax und Semantik und erleichtern somit die Entwicklung eindeutiger, konsistenter und vollständiger Systembeschreibungen. Formale Analyse- und Verifikationstechniken erlauben die Überprüfung von Systemen auf Erfüllbarkeit, korrekte Implementierungen und weitere spezifische Eigenschaften.

Durch formale Methoden lassen sich Aussagen über die Leistung, Echtzeitfähigkeit und Funktionalität von Systemen machen (vgl. [Kru90a]). Im folgenden beschränken wir uns jedoch auf Methoden, die das funktionale Systemverhalten beschreiben. Wir geben in Abschnitt 3.1 zunächst eine Einführung in formale Methoden zur funktionalen Systembeschreibung und -analyse. Danach geben wir in Abschnitt 3.2 eine kurze Übersicht über die am häufigsten für den Protokollentwurf verwendeten Techniken. Anschließend führen wir in die Spezifikationstechnik TLA [Lam94] sowie in die zugehörigen Spezifikationssprachen TLA$^+$ [Lam95] und tTLA$^+$ [Mes93] ein. Wir schließen das Kapitel mit einer Übersicht über Werkzeuge zur Modellierung und Analyse von Systemen auf der Basis von TLA ab.

3.1 Grundlagen

Wing definiert formale Methoden folgendermaßen:

> „A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides the means of precisely defining notions like consistency and completeness and, more relevantly, specification, implementation, and correctness.“ (Zitat aus [Win90])

Im weiteren werden wir uns auf Methoden konzentrieren, die auf operationalen Systemmodellen basieren. Krumm [Kru90a] versteht darunter Modelle, in denen der reale zeitliche Ablauf von Systemen, der sich aus Operationen zusammensetzt, eine direkte Entsprechung finden kann. Operationale Modelle stellen sicher, daß die Ergebnisse der Beschreibungen und Analysen der Modellwelt auf die realen Systeme übertragen werden können.

Seitdem am Ende der Sechziger Jahre Software als kritischer Faktor in der Computerentwicklung erkannt wurde (vgl. [Knu73, Gib94]), entstand eine Vielzahl an Ansätzen und Entwürfen funktionaler und formaler Methoden. Zu ihrer Klassifikation dienen im wesentlichen die fünf Aspekte Anwendungsgebiet, Modellbildung, Notation, Systembildung und Verifikationsmethode (vgl. auch [HK92]), die wir im folgenden vorstellen werden.

3.1.1 Anwendungsgebiet

Formale Methoden können auf allen Gebieten des Systementwurfs von der Beschreibung eines Pflichtenhefts über die Systementwicklung, die Implementierung bis zur Wartung laufender Systeme eingesetzt werden (vgl. [Win90]). Die in der Systementwicklung eingesetzten Analysemethoden kann man in die Bereiche Verifikation, Testen und Transformation einteilen.

Unter formaler Verifikation versteht man die Überprüfung der Gültigkeit eines bereits abgeschlossenen Entwicklungsschrittes unter Verwendung geeigneter Modellwelten, Notationen und Analysemethoden (vgl. [IEE90]). Ziel einer Verifikation kann die Überprüfung des entwickelten Systems auf allgemeine Systemeigenschaften wie Terminierung, Fehlerfreiheit und Verklemmungsfreiheit oder spezifische Systemeigenschaften wie eine korrekte Relation von Systemeingaben und -ausgaben sein (vgl. [AO91b]). Ein weiteres Verifikationziel ist der direkte Vergleich der beiden Systeme, die am Ausgang bzw. am Ende eines Entwicklungsschrittes stehen. Ergebnis eines solchen Vergleichs kann eine Relation zwischen den Systemen wie „System Φ implementiert/ist eine Verfeinerung von System Ψ“ oder „Φ und Ψ sind semantisch äquivalent“ sein. Diese Relationen müssen dabei nicht unbedingt das vollständige Systemverhalten beschreiben, sondern können sich auch nur auf Abstraktionen wie das nach außen sichtbare Systemverhalten beziehen. Die Systemeigenschaften unterscheiden sich nach Safetyeigenschaften und Livenesseigenschaften (vgl. [AS85]). Safetyeigenschaften beschreiben, daß in einem System bestimmte unerwünschte Systemschritte nicht auftreten (vgl. [Lam77]). Als Beispiel betrachten wir dazu zwei miteinander konkurrierende Rechnerprozesse, die sich um den Eintritt in einen kritischen Abschnitt bewerben, in dem sich gleichzeitig nur ein Prozeß aufhalten darf. Die Eigenschaft, daß ein Prozeß nur in den kritischen Abschnitt eintritt, wenn sich der andere dort nicht befindet, ist eine typische Safetyeigenschaft. Dagegen sagen Livenesseigenschaften aus, daß Systemschritte, die immer wieder erfolgen dürfen, irgendwann tatsächlich eintreten. Die Eigenschaft, daß ein Prozeß, der auf den Eintritt in den kritischen Abschnitt wartet, nicht immer von seinem Konkurrenten verdrängt wird, sondern irgendwann tatsächlich berücksichtigt wird, ist eine Livenesseigenschaft.

Unter funktionalem Testen versteht man die Überprüfung lauffähiger Systeme auf bestimmte geforderte Systemeigenschaften. Unter typischen oder auch grenznahen Randbedin-

gungen wird durch Systemabläufe untersucht, ob bestimmte Systemeingaben zu den erwarteten Systemausgaben führen (vgl. [Fai85]).

Auf Transformation basierende Entwicklungstechniken unterstützen die Systembildung durch korrektheitserhaltende Transformationsregeln. Die Systementwicklung wird in viele kleine Entwicklungsschritte zerlegt, in denen jeweils eine Transformationsregel angewendet wird. Während Transformationen im Bereich der Übersetzung mechanisch eingesetzt werden, ist in der „Source-Code"-Transformation die kreative Auswahl von Transformationsregeln durch den Systementwickler notwendig. Die Auswahl der Regeln hängt dabei von semantischen Systembedingungen ab, die nachgeprüft werden müssen (vgl. auch [Bar85, Bar88, BDD+92, DW92, vES91]). „Source-Code"-Transformation kann somit als durch Verifikation gestützte Systementwicklung in sehr kleinen Entwicklungsschritten aufgefaßt werden.

3.1.2 Modellbildung

Das funktionale Verhalten nebenläufiger Systeme kann durch direkte nebenläufige Modelle oder durch „Interleaving"-Modelle abgebildet werden (vgl. [Lam82]). In direkten Modellen wird die Nebenläufigkeit von Ereignissen durch eine Halbordnung beschrieben. In „Interleaving"-Modellen wird das funktionale Verhalten durch eine Menge atomarer Ereignisse beschrieben, deren zeitlicher Ablauf totalgeordnet ist. Dabei kann die Reihenfolge zueinander nebenläufiger Ereignisse indeterministisch gewählt werden.

In Hinsicht auf die Ausführung von Ereignissen innerhalb des Systems unterscheidet man die drei Sichten der typisierten Ereignisse, der kommunizierenden Agenten und der Zustandstransitionsmodelle. In der ersten Sicht besteht ein System aus einfachen Agenten, die miteinander durch Austausch von Ereignissen eines bestimmten Typs kooperieren. Das Modell definiert die Ereignistypen und auch die Halbordnung bzw. gegenseitige Abhängigkeit der Ereignisse. In der Sicht der kommunizierenden Agenten setzt sich ein System aus mehreren komplexeren Agenten zusammen, die intern im wesentlichen sequentielle Abläufe besitzen und mit anderen Agenten ebenfalls durch Transfer von Ereignissen kommunizieren. Die Koordination kann durch direkte „Rendez-Vous"-Kopplung oder durch gepufferten Ereignisaustausch erfolgen. Die Zustandstransitionsmodelle beschreiben Systeme durch Folgen globaler Systemzustände, die durch diskrete Transitionen verändert werden. Das Modell sieht a priori keine zueinander disjunkten Agenten vor.

Modelle kann man weiterhin in Bezug auf ihre Abstraktionsfähigkeit differenzieren. Außer Modellen, die keine Abstraktionen zulassen, existieren Modelle, die eine abstrahierende makroskopische Systemsicht unterstützen, in der Systemdetails ausgeblendet sind. Eine dritte Klasse erlaubt die Definition von Modellrelationen, in denen detaillierte Modelle zu Modellen in Beziehung gesetzt werden, die nur abstrakte Sichten von ihnen beschreiben (vgl. [LS84]).

3.1.3 Notation

Formale Spezifikationssprachen lassen sich in konstruktive und in deskriptive Sprachen aufteilen. Konstruktive Sprachen ermöglichen direkte Modellbeschreibungen, also Quasi-Implementierungen eines modellierten Systems. Deskriptive Sprachen hingegen erlauben die Modellierung von Eigenschaften, die ein System erfüllen soll. Mithin werden keine einzelnen Modelle, sondern Klassen aller Modelle spezifiziert, die die modellierten Eigenschaften erfüllen.

Beispiele für konstruktive Spezifikationssprachen sind erweiterte kommunizierende endliche Zustandsautomaten, kommunizierende sequentielle Prozeßsysteme und Petrinetze. Er-

weiterte kommunizierende endliche Zustandsautomaten (Extended Communicating Finite State Machines, ECFSMs) [Mer76, Mer79] wurden entwickelt, da realistische Protokolle mit einfachen Mealy-Automaten (vgl. [HU79]) nur stark abstrahiert dargestellt werden können. ECFSMs erlauben eine modulare Beschreibung von Zuständen durch Variablen und Parameter. Der Zugriff auf die Variablen wird durch Bedingungen und Zuweisungen modelliert, die den Transitionen zugeordnet sind. Zur Systembildung stellen ECFSMs „Rendez-Vous"-Kopplungsmechanismen, geteilte Variablen [LM91] oder „Message-passing"-Puffer [BZ83] zur Verfügung (siehe Abschnitt 3.1.4). ECFSMs mit „Message-passing"-Puffern bilden die Grundlage der von den Standardisierungsorganisationen ITU bzw. ISO genormten formalen Beschreibungstechniken (Formal Description Techniques, FDTs) SDL [ITU93] und ESTELLE [ISO89a] (siehe Abschnitt 3.2.1). Für die beiden Techniken sind pseudo-programmiersprachliche Spezifikationssprachen definiert. SDL wird zusätzlich durch eine graphische Notation unterstützt.

Kommunizierende sequentielle Prozeßsysteme (CSP) [Hoa85] basieren auf Zustandstransitionssystemen mit markierten Transitionen und „Rendez-Vous"-Kopplungsmechanismen (siehe Abschnitt 3.1.4). Sie bilden die Basis für die auf Prozeßalgebren basierenden Spezifikationstechniken wie CCS [Mil80] und der ebenfalls von der ISO standardisierten FDT LOTOS [ISO89b] (siehe Abschnitt 3.2.1). In diesen Sprachen wird die Systembildung durch Operatoren zur Modellierung von unabhängiger, abhängiger bzw. allgemeiner Parallelität und durch einen Operator zur Verdeckung von Ereignissen unterstützt (vgl. [Hog89]).

Petrinetze [Mer76, BWH86, Bil91] eignen sich ebenfalls zur formalen Beschreibung verteilter Systeme. Sie erlauben die sehr direkte Modellierung von Nebenläufigkeit. Durch Produktnetze [BOP87], in denen Transitionen bestimmten Agenten oder Schnittstellen zugeordnet werden, können aus Agenten zusammengesetzte Systeme dargestellt werden. Petrinetze können graphisch oder textuell durch Tabellen dargestellt werden.

Konstruktive Sprachen beschreiben im wesentlichen Safetyeigenschaften und nur sehr eingeschränkte Livenesseigenschaften. Es existieren allerdings auch Zustandstransitionsmodelle, die die Spezifikation von Livenesseigenschaften unterstützen (z.B. Büchi-Automaten [Eil74], $\forall$-Automaten [MP87]).

Deskriptive Modellspezifikationen können gut in logischen Sprachen beschrieben werden. Neben klassischen logischen Sprachen wie der Aussagenlogik und der Prädikatenlogik werden in den letzten 15 Jahren vermehrt modale Logiken, insbesondere temporale Logiken, zur Systemspezifikation eingesetzt (vgl. [Lam80, Pnu77]). Sie ermöglichen eine einfache Beschreibung von unendlich langen Zustandsbäumen oder Zustandsfolgen. Einzelne Zustände werden durch Formeln in klassischer Aussagen- oder Prädikatenlogik beschrieben, während zur Spezifikation von Zustandsbäumen oder Zustandsketten zusätzliche modale Operatoren verwendet werden. Temporale Logiken lassen sich in Linearzeit-Temporallogiken, Baum-Temporallogiken und Halbordnungs-Temporallogiken zerlegen. Linearzeit-Temporallogiken [Lam94, MP92a] modellieren die Zukunft als Menge von linearen Zustandsketten und unterstützen somit „Interleaving"-Modelle. Baum-Temporallogiken [CES86, EH86] modellieren die Zukunft dagegen durch einen einzigen Zustandsbaum. Halbordnungs-Temporallogiken [Wol89] unterstützen die Beschreibung direkter Nebenläufigkeit durch einen temporalen Halbordnungsoperator.

Aus der Sicht des Systementwicklers sind deskriptive Sprachen in einem frühen Entwurfsstadium sinnvoll, da sie die Beschreibung der geforderten Systemeigenschaften zulassen (extensionale Beschreibungen), ohne schon Quasi-Implementierungen mit eventuell vorweg-

genommenen Entwurfsentscheidungen zu modellieren. Im weiteren Verlauf der Systementwicklung werden die Systembeschreibungen immer detaillierter und können leichter durch konstruktive Sprachen spezifiziert werden (intensionale Beschreibungen). Um für die gesamte Systementwicklung einheitliche Methoden zu unterstützen, wurde versucht, deskriptive und konstruktive Techniken miteinander zu kombinieren. Hennessy und Milner schlagen in [HM85] eine Methode vor, in der die Hennessy-Milner-Logik, eine modale Logik, die den Beweis der Äquivalenz des sichtbaren Verhaltens von nebenläufigen Systemen zuläßt, mit der Spezifikationsmethode CCS (siehe Abschnitt 3.2) kombiniert. Broy et al. [BDD+92, DW92] unterstützen mit ihrer auf Konstruktion basierenden Entwicklungsmethode FOCUS den Entwurf abstrakter Systeme durch prädikatenlogische Formeln, den Entwurf detaillierter Systeme durch Datenstrom-verarbeitende Funktionen, die über Ereignis-Warteschlangen kommunizieren, und die Implementierung durch funktionale und imperative Programmiersprachen.

Diese kombinierten Techniken haben jedoch den Nachteil, daß mindestens zwei verschiedene Techniken verwendet werden müssen und eine Umsetzung der Ergebnisse notwendig ist. Hier bietet eine Spezifikationstechnik wie Lamports TLA [Lam94] Vorteile, die sowohl deskriptive als auch konstruktive Systemspezifikationen zuläßt. TLA erlaubt die Modellierung von Systemeigenschaften durch logische Formeln. Man kann Systeme aber auch in einer kanonischen Form beschreiben, in der die Safetyeigenschaften durch Aktionen (Mengen von Transitionen) beschrieben werden. Die kanonische Form entspricht dann einer Beschreibung eines konstruktiven Zustandstransitionsmodells. TLA wird in Abschnitt 3.3 vorgestellt.

3.1.4 Systembildung

Da verteilte Systeme sich aus mehreren räumlich verteilten Instanzen zusammensetzen, ist eine formale Spezifikationsmethode sinnvoll, in der die Instanzen als zueinander separate Agenten modelliert und die Agenten schließlich zu einem Systemmodell zusammengesetzt werden können. Lamport und Lynch [LL90] unterscheiden drei Modellierungsmethoden für verteilte Systeme. In „Message-passing"-Modellen kommunizieren Agenten durch Transfer von Nachrichten über Kanäle. Die Kanäle besitzen Warteschlangen, in denen die Nachrichten zwischengespeichert werden, so daß die Kommunikation asynchron verläuft. Durch spezielle Warteschlangeneigenschaften wie begrenzte Kapazität, Zugriffsstrategie oder Nachrichtenverfälschung können Eigenschaften des Kommunikationskanals abgebildet werden. Eine andere Modellierungsmethode verwendet von mehreren Instanzen gemeinsam genutzte Variablen („Shared Variables") zur Kommunikation, die entweder durch spezielle Synchronisationsprimitive wie Semaphore oder Monitore synchronisiert werden oder von beliebig vielen Agenten gelesen, aber nur von einem ausgezeichneten Agenten beschrieben werden. Diese Methode ermöglicht auch eine asynchrone Kommunikation von Agenten. Die Methode der „Rendez-Vous"-Kopplung sieht dagegen eine synchrone Kommunikation vor. Bei der Übertragung von Nachrichten treten das Sendeereignis und das Empfangsereignis zueinander simultan auf.

Ein anderes Klassifikationskriterium für die Systembildung ist die Struktur von Systemspezifikationen. Vissers et al. [VSvS88, VSvS91] schlagen vier Spezifikationsstile vor (monolithischer, constraintorientierter, ressourcenorientierter, zustandsorientierter Stil). Der monolithische und der constraintorientierte Spezifikationsstil eignen sich insbesondere für extensionale Beschreibungen. Der monolithische Stil beschreibt das nach außen sichtbare Systemverhalten durch eine Menge von Interaktionsfolgen, ohne auf Implementierungsaspekte wie die interne Struktur oder die räumlich-verteilte Anordnung von Komponenten einzuge-

hen. Im constraintorientierten Stil wird ein System durch einzelne Eigenschaften beschrieben, die separat modelliert werden und zu einer Gesamtsystembeschreibung komponiert werden (vgl. [Bri90]). Constraintorientierte Spezifikationen können sowohl mit asynchronen als auch synchronen Modellen beschrieben werden. Nach unserer Meinung sind synchrone Modelle vorzuziehen, da Funktionen, die unterschiedliche Eigenschaften modellieren, relativ stark miteinander interagieren, so daß in asynchronen Techniken ein hoher Koordinationsaufwand notwendig ist. Zum Beispiel soll ein Kommunikationsdienst modelliert werden, in dem eine von einem Dienstnehmer gesendete Nachricht zunächst durch mehrere Funktionen, die als Constraints modelliert sind, modifiziert wird. Wenn alle Funktionen terminiert sind, wird die modifizierte Nachricht an einen anderen Dienstnehmer ausgeliefert. In einem synchronen Modell kann man die Auslieferung durch eine Interaktion modellieren, in der in jedem Constraint ein Ereignis schaltet. Dabei darf das Ereignis nur schalten, wenn die durch das Constraint beschriebene Funktion terminiert ist. In einem asynchronen Modell muß durch einen zusätzlichen Mechanismus (z.B. dem Zwei-Phasen-Commit) ermittelt werden, ob alle Funktionen terminiert sind und damit die Auslieferung erfolgen darf.

Intensionale Beschreibungen werden durch den zustandsorientierten und den ressourcenorientierten Spezifikationsstil unterstützt. Im zustandsorientierten Stil wird ein System durch ein Zustandstransitionsmodell beschrieben. Der ressourcenorientierte Stil erlaubt eine Beschreibung von Systemen unter expliziter Berücksichtigung der Systemressourcen, deren äußeres Verhalten durch eigenständige Spezifikationen modelliert wird. Die Ressourcenspezifikationen können in einem beliebigen Spezifikationsstil modelliert werden. Es ist also möglich, constraintorientierte und ressourcenorientierte Spezifikationen zu kombinieren. Bei der Modellierung von Kommunikationsprotokollen und -diensten ist eine solche Kombination sinnvoll. Man kann Dienste constraintorientiert durch Kopplung von Spezifikationen ihrer Diensteigenschaften beschreiben. Durch den kombinierten Spezifikationsstil kann man in Protokollspezifikationen außer den Protokollinstanzen auch die Protokollmechanismen beschreiben, aus denen sich die Protokollinstanzen zusammensetzen. Durch Komposition einer meist geringen Anzahl an Protokollmechanismen kann man die Spezifikation einer Protokollfunktion erzeugen, die eine einzelne Diensteigenschaft realisiert. Somit bildet der kombinierte Stil eine Modellierungsbasis für die Zerlegung von Protokollverifikationen in kleinere Schritte, in denen jeweils bewiesen wird, daß eine Diensteigenschaft durch eine Protokollfunktion implementiert wird.

3.1.5 Verifikationsmethoden

Die formale Verifikation von Spezifikationen kann modellbasiert, syntaktisch oder hybrid erfolgen. Bei der modellbasierten Verifikation werden allgemeine mathematische bzw. logische Beweistechniken auf die Strukturen der Modellwelt (z.B. Zustandsfolgen) angewendet. Beispiele für diese Methode ist die Erreichbarkeitsgraphanalyse, die Analyse von Markierungsgraphen beschränkter Petrinetze, die Produktautomatenbildung und die Invarianten-Analyse von Petrinetzen.

Die Erreichbarkeitsgraphanalyse [RW82, vBo82, Wes78, ZWR$^+$80] ist eine auf „Interleaving“-Modellen aufgebaute Technik für Zustandstransitionsmodelle mit endlich vielen Zuständen. Ausgehend von initialen Systemzuständen wird ein Graph aller erreichbaren Systemzustände als Knoten und aller erreichbaren Zustandsübergänge als Kanten vollautomatisch generiert. Die Graphen können zum einen auf allgemeine Systemeigenschaften wie

Vollständigkeit, Verklemmungsfreiheit, garantierten Fortschritt, Terminierung, Abwesenheit nicht verwendeter Zustandsübergänge oder Beschränkung des Nachrichtenflusses überprüft werden (vgl. [Rud82, Rud86]). Zum anderen können zwei Erreichbarkeitsgraphen auf die Äquivalenz des nach außen hin sichtbaren Verhaltens untersucht werden (vgl. [Sun83]). Das Problem der aufgrund des „Interleaving“ auftretenden Zustandsexplosion kann durch Modellvereinfachung ohne Veränderung des nach außen sichtbaren Verhaltens [LS84, Och94] und durch Strukturierung der Erreichbarkeitsgraphen [Kru90b] abgemildert werden. Bei komponierten Systemen können Verhaltenseinschränkungen, die sich aus den Eigenschaften der Agentenkopplung ergeben, zu einer Reduktion der Erreichbarkeitsgraphen eingesetzt werden [Chi89, LM91, SLU89, YK90]. Zu einem spürbaren Fortschritt führte der Supertrace-Algorithmus von Holzmann [Hol88, Hol90] und seine Erweiterungen [WL93, Hol95], der mit Hilfe von Bit-Hashing eine sehr effektive Zustandsspeicherung erlaubt und damit die Erzeugung und Verwaltung von Erreichbarkeitsgraphen mit 10^7 Zuständen auf heutigen Standard-Workstations ermöglicht. Da die meisten Kommunikationsprotokolle zwischen 10^5 und 10^7 Zustände besitzen (vgl. [Hol87]), ist Erreichbarkeitsanalyse eine praktikable Methode in der Protokollverifikation.

Da Markierungsgraphen von beschränkten Petrinetzen einem endlichen Zustandstransitionssystem entsprechen, läßt sich die Erreichbarkeitsanalyse auch in dieser Modellwelt einsetzen (vgl. [CP95, God90, GP93, JK90, Kem96, Val89, WG93]). Bei der Invarianten-Analyse werden Petrinetze vollautomatisch auf das Vorhandensein bestimmter invarianter Markenbelegungen untersucht (vgl. [MS82]).

In der syntaktischen Systemverifikation bildet die Syntax der Spezifikationssprache die formale Basis. Durch Axiome und Deduktionsregeln wird die Spezifikationssprache zu einem Kalkül erweitert, das rein syntaktische Schlußfolgerungen erlaubt. Im Gegensatz zur Korrektheit eines Kalküls ist dessen Vollständigkeit nicht immer gegeben (Linearzeit-Temporallogikkalküle sind sogar inhärent unvollständig; vgl. [Wol82]). Allerdings können unvollständige Kalküle in einem praktischen Sinn vollständig sein. Das ist dann der Fall, wenn sie alle Beweise unterstützen, die in ihrem Einsatzgebiet auftreten. Beispiele für syntaktisches Schließen sind Prozeßalgebren und temporallogische Kalküle.

In Prozeßalgebren erfolgt die Verifikation durch Termersetzung auf der Basis von Äquivalenzaxiomen [Mil80, Mil83]. Die Axiome definieren dabei Äquivalenz- oder Kongruenzrelationen auf den Operatoren der Prozeßalgebra und bilden die Termersetzungsregeln. Die Technik unterstützt die Vereinfachung von Systemen und die Entscheidung, ob zwei Systeme zueinander äquivalent sind. Für Systeme mit endlichem Zustandsraum existieren vollautomatische Werkzeuge für die Systemvereinfachung [AW91, MV91] und für den Äquivalenzvergleich [CGS91, MM91]. Weiterhin kann die Systemkompositionalität ausgenutzt werden, da ein Teil der Äquivalenzaxiome sich nur auf einzelne Agenten oder Subsysteme bezieht (vgl. [MC81, BLB93, LS91b]).

Temporallogisches Schließen [CM88, Lam94, MP92a] basiert auf logischer Deduktion im Prädikatenlogikkalkül erster Ordnung und auf weiteren temporallogischen Deduktionsregeln. Als Werkzeuge werden interaktive Theorembeweiser eingesetzt [EGL92, GGH90, Gol90, vWL92, WWM$^+$90]. Durch Theorembeweiser-Frontend-Werkzeuge kann die Überführung temporaler Formeln in prädikatenlogische Formeln unterstützt werden [EHK93, ORS92]. Systemkompositionalität wird ausgenutzt, indem Verifikationsbeweise in einfachere Beweisschritte zerlegt werden, in denen nur einzelne Systemagenten oder Subsysteme referenziert werden (vgl. [AL93a, MP92b]).

Vorteile der modellbasierten Verifikation ist die Vollständigkeit der Technik und die gute, meist vollautomatische Werkzeugunterstützung. Die syntaktische Verifikation besticht hingegen durch die Einfachheit und die Übersichtlichkeit der Deduktionsregeln (zumindestens in Logikkalkülen). Um diese Vorteile verbinden zu können, setzt man vermehrt hybride Methoden wie Model Checking und Modellkonstruktion ein (vgl. [HV91]).

Im Model Checking wird ein Modell (z.B. eine konstruktive Spezifikation oder eine Implementierung) mit einer Eigenschaftsbeschreibung (z.B. einer temporallogischen Formel) verglichen. Durch ein Werkzeug wird untersucht, ob das Modell die beschriebene Eigenschaft erfüllt. Ein Modell mit einem endlichen Zustandsraum wird durch seinen Erreichbarkeitsgraph repräsentiert, dessen Zustände bzw. Zustandsfolgen auf Einhaltung der Eigenschaft untersucht werden (vgl. [ACD91, BCM+92, CES86, CGL94]). Für bestimmte Logiken kann man Entscheidungsprozeduren mit im Vergleich zur Zustandsanzahl des Erreichbarkeitsgraph polynomiellem Aufwand entwickeln (vgl. [BE91]). Weiterhin kann man „On-the-fly"-Model Checker entwickeln, in denen für die Überprüfung von Safetyeigenschaften nur einzelne Zustände (vgl. [FM91, JJ91]) und von Livenesseigenschaften nur Teilgraphen (vgl. [CGL94, CVWY90, DHWT91, GPVW95, HKKS94]) zu einem Zeitpunkt gespeichert sein müssen.

Im Vergleich zum Model Checking geht die Modellkonstruktion davon aus, eine Spezifikation in ein Modell zu überführen (vgl. [Bar85, Bar88]), das die Eigenschaften der Spezifikation erfüllt. Die Modellentwicklung auf abstrakter Ebene (im Gegensatz zur Übersetzung von Programmen, die auch eine Modellkonstruktion ist) kann im allgemeinen nur interaktiv durch Transformationsschritte erfolgen (vgl. [BDD+92, Fre87]). Für spezielle Anwendungsfelder, zum Beispiel die Spezifikation bestimmter abstrakter Datentypen, kann die Modellkonstruktion automatisiert werden (vgl. [CR90]).

3.2 Formale Protokollentwicklung

Aus der Vielzahl dieser formalen Spezifikations- und Analysemethoden haben sich einige für die Entwicklung von Kommunikationsprotokollen durchgesetzt, die wir kurz vorstellen werden. Anschließend geben wir eine Übersicht über die im Moment noch nicht zahlreichen Arbeiten zur formalen Entwicklung von Hochleistungsprotokollen. In Abschnitt 3.2.3 ziehen wir daraus ein Resümee und begründen, warum uns Lamports TLA [Lam94] als gut geeignet zur Modellierung von Hochleistungsprotokollen erscheint.

3.2.1 Formale Beschreibungs- und Analysetechniken

Der Einsatz formaler Methoden zur Beschreibung und insbesondere zur Analyse realer verteilter Systeme ist immer noch sehr gering. Außer semiformalen Beschreibungstechniken wie Zustandstabellen oder Weg-/Zeitdiagrammen finden die drei standardisierten formalen Beschreibungstechniken SDL, ESTELLE und LOTOS noch die weiteste Verbreitung. Insbesondere die von der Poststandardisierungsorganisation CCITT[1] (Comité Consultatif International des Télégraphique et Téléphonique) genormte Spezifikationssprache SDL (Specification and Description Language) [ITU93] wird häufig eingesetzt, da sie zum einen von den Postgesellschaften gefördert wird und zum anderen eine recht einfache graphische Notation

[1] Heute International Telecommunications Union (ITU)

SDL/GR (SDL Graphical Representation) zur Verfügung stellt. Das Basiskonzept in SDL sind durch erweiterte endliche Automaten modellierte Prozesse, die zueinander gleichrangig sind. Die Automatenzustände werden durch typisierte Variablen beschrieben. Die Datentypen entsprechen dem abstrakten Datentypkonzept nach ACT ONE [EM85]. Die Kooperation der Prozesse erfolgt über typisierte FIFO-Warteschlangen von unbegrenzter Kapazität und entspricht somit dem „Message-passing"-Konzept (siehe Abschnitt 3.1.4). SDL unterstützt ein auf Zeitgebern und einem abstrakten Zeit-Datentyp basierendes Realzeitkonzept. Prozesse und Warteschlangen können zu Blöcken zusammengesetzt werden, die mit ihrer Umgebung über Kanäle kooperieren. Aus Blöcken können hierarchische Strukturen gebildet werden.

Eine zu SDL sehr ähnliche Spezifikationssprache ist ESTELLE [ISO89a], die von der ISO genormt wurde. ESTELLE basiert ebenfalls auf erweiterten endlichen Automaten und auf typisierten FIFO-Warteschlangen. Als Beschreibungssprache von Variablen und Transitionen wird eine Obermenge der Programmiersprache PASCAL [JW78] verwendet. Prozesse, die in ESTELLE Module genannt werden, können selbst Moduldefinitionen enthalten, so daß hierarchische Spezifikationen möglich sind.

Zur Analyse von durch erweiterte endliche Automaten modellierten Systemen werden Erreichbarkeitsanalyse und Model Checking eingesetzt. Aus pragmatischen Gründen werden Spezifikationen manchmal direkt in Eingabesprachen von Analysewerkzeugen verfaßt. Beispiele dafür sind PROMELA als Eingabesprache des Analysewerkzeugs SPIN [Hol91, Hol93], das auf dem Supertrace-Algorithmus basiert, und S/R als Notation des Synthesewerkzeugs COSPAN [HK90].

Die ebenfalls von der ISO standardisierte Beschreibungssprache LOTOS [ISO89b] verfolgt dagegen das Konzept der kommunizierenden sequentiellen Prozeßsysteme [Hoa85], in denen sich der Systemzustand aus der Folge der bisher eingetretenen Ereignisse ergibt. Das Basiskonzept sind Prozesse, deren beobachtbares Verhalten durch Ereignisse definiert ist. Das Prozeßverhalten wird durch Bäume erlaubter Ereignisse beschrieben, wobei auch interne, nach außen nicht sichbare Ereignisse auftreten können. In der Erweiterung Extended-LOTOS werden Ereignisse durch abstrakte Datentypen nach ACT ONE [EM85] typisiert. Durch „Rendez-Vous"-Kopplung von Ereignissen kann man Prozesse zueinander synchron komponieren. LOTOS sieht mehrere Operatoren zur Beschreibung der Parallelität von zwei Prozessen vor. Bei der unabhängigen Komposition sind die Prozesse vollständig nebenläufig und führen niemals gleichzeitig ein Ereignis aus. In der parallelen Komposition dagegen müssen alle Ereignisse der beiden Prozesse zueinander synchron ausgeführt werden. In der allgemeinen parallelen Komposition kann ausgewählt werden, welche Ereignisse synchron ausgeführt werden sollen. Durch einen Verdeckungsoperator kann das Ausführen bestimmter Ereignisse für die Umgebung von Prozessen unsichtbar gemacht werden. LOTOS basiert auf der Spezifikationssprache CCS [Mil80]. Die Verifikation in LOTOS und CCS erfolgt durch Verwendung von Prozeßalgebren (siehe Abschnitt 3.1.5).

Eine weitere Spezifikationsmethode sind Petrinetze [Mer76, Bil91]. Das sind bipartite gerichtete Graphen, deren Knoten als Stellen und Transitionen bezeichnet werden. Stellen werden markiert durch eine natürliche Zahl, die Anzahl der sogenannten Marken. An den Stellen, die mit einer Transition verbunden sind, kann eine Veränderung der Marken erfolgen (die Transition „feuert"). Wenn an allen Stellen, zu denen die Transition eine Eingangskante besitzt, mindestens eine Marke vorhanden ist, wird beim „Feuern" je eine Marke entfernt und allen Stellen, zu denen die Transition eine Ausgangskante besitzt, eine Marke hinzugefügt. Am Anfang wird eine Markenbelegung vorgegeben. Es existieren diverse Varianten von Petri-

netzen wie Produktnetze oder Zeit-Petrinetze. Formale Analyse in Petrinetzen erfolgt durch Erreichbarkeits- oder Invariantenanalyse (siehe Abschnitt 3.1.5).

In der stromorientierten formalen Entwicklung werden detaillierte Systeme aus abstrakten Spezifikationen durch korrektheitserhaltende Transformationen gebildet. Dabei können wie in der Entwurfsmethode FOCUS [BDD+92] unterschiedliche Notationen für unterschiedliche Abstraktionsebenen verwendet werden, so daß Transformationen Sprachübersetzungsschritte enthalten.

Auf temporalen Logikkalkülen basierende Systeme wie Mannas und Pnuelis „Temporal Logic of Reactive and Concurrent Systems“ [MP92a] oder Lamports TLA [Lam94] werden bisher nur vereinzelt zur Spezifikation und Verifikation verteilter Systeme eingesetzt (vgl. [BC95]). TLA und seine Spezifikationssprachen TLA$^+$ und tTLA$^+$ werden in den Abschnitten 3.3 und 3.4 erklärt. Mannas und Pnuelis Logik kennt im Gegensatz zu TLA kein Aktionskonzept und verwendet stattdessen eine komplexere Logik, in der temporale Vergangenheits- und Folgezustandsoperatoren verwendet werden.

3.2.2 Anwendung formaler Methoden auf Hochleistungstransportprotokolle

Auf der Basis der oben genannten Spezifikationstechniken wurden vereinzelt Hochleistungstransportprotokolle formal modelliert. Cheung und Atwood [CA90] haben das Protokoll XTP (siehe Abschnitt 2.4.4) in ESTELLE spezifiziert. Sie haben eine einfache constraintorientierte Aufteilung des Protokolls erzielt, indem die Sende- und Empfangsströme und die Nutzer- bzw. Basismedium-orientierten Funktionen separaten Modulen zugeordnet wurden. Die Spezifikation bildete die Ausgangbasis einer XTP-Implementierung in PASCAL. Eine formale Verifikation der Spezifikation gegenüber einer abstrakteren Anforderungsdefinition fand nicht statt. Mit dem Analysewerkzeug VALIRA [VL88] wurde eine partielle Erreichbarkeitsanalyse vorgenommen. Um die Größe des Erreichbarkeitsgraphen zu beschränken, wurden bestimmte Datentransferfehler ausgeschlossen und eingeschränkte Nebenläufigkeit angenommen. Durch die Erreichbarkeitsanalyse sollten der Datentransfer zwischen den Protokollmodulen ermittelt werden. Die Überprüfung des Protokolls auf Fehler war nur nebensächlich. Einen ähnlichen Ansatz verfolgt Catrina [Cat95], der die jüngste Version 4.0 von XTP ebenfalls in ESTELLE spezifiziert und mit Hilfe des Entwicklungswerkzeugs EDT (ESTELLE Development Toolset; vgl. [Bud92]) simuliert und per partieller Erreichbarkeitsanalyse auf bestimmte kausale Zusammenhänge von Protokollereignissen überprüft hat. Heinrichs [Hei92] verwendet Petrinetze zur Spezifikation von XTP, die er jedoch nicht analysiert, sondern als Ausgangspunkt für eine parallele Implementierung einsetzt.

Lundy et al. [LM92, LT94] verwenden eine auf endlichen Automaten basierende Modellwelt, in dem die Agenten über geteilte Variablen kommunizieren (vgl. [LM91]). Sie spezifizieren damit das Protokoll SNR (siehe Abschnitt 2.4.5), für das sie einen Simplex-Transportkanal mit Verbindungsmanagement und Datentransfer modellieren. Dazu werden constraintorientierte Automaten für die Verbindungsverwaltung, den Datentransfer und den Transfer von Steuerinformation spezifiziert. Durch Erreichbarkeitsanalyse werden allgemeine Systemeigenschaften (z.B. garantierter Fortschritt, nicht ausgeführter Code) überprüft. Dabei werden aus Gründen der Zustandsexplosion nur Abstraktionen der Spezifikationen formal analysiert. Um dennoch Aussagen über die Systemsicherheit zu erhalten, wurde das Gesamtsystem simuliert.

Eine ähnlich geartete Spezifikation und Analyse von PATROCLOS (siehe Abschnitt 2.4.6)

führt Stocker mit dem Analysewerkzeug SPIN [Hol91, Hol93] durch. Dazu wurde PATROCLOS in der Eingabesprache PROMELA modelliert, wobei die PATROCLOS-Automaten durch PROMELA-Prozesse beschrieben werden. Mit SPIN können Spezifikationen sowohl simuliert als auch auf allgemeine Systemeigenschaften überprüft werden. Stocker verzichtete jedoch aus Komplexitätsgründen auf die formale Überprüfung und simulierte nur einige interessante Protokollszenarien.

La Porta und Schwarz [LS94] analysieren das Protokoll MSP (siehe Abschnitt 2.4.7) mit dem Analyse- und Synthesewerkzeug COSPAN [HK90]. Zuerst wurde eine sehr abstrakte Beschreibung von MSP in der auf Zustandstransitionsautomaten basierenden Eingabesprache S/R angefertigt und per Model Checking auf einige Safety- und Livenesseigenschaften überprüft. Der Schwerpunkt dieser Überprüfungen war dabei die Interaktion zwischen den verschiedenen MSP-Prozessen. Anschließend wurde eine detailliertere Protokollspezifikation erstellt. Auf die von COSPAN vorgesehene Verifikation der Korrektheit dieser Spezifikation gegenüber der abstrakten Beschreibung wurde jedoch aus uns nicht bekannten Gründen verzichtet.

Temporallogik verwenden Barnard und Crosby [BC95], die die Logikeingabesprache U-TLT zur Spezifikation des „User-Master-Interface“ von ATM (siehe Abschnitt 2.4.1) verwenden. Durch ein Konsistenz-Überprüfungswerkzeug wird die Spezifikation zunächst auf Widersprüche untersucht. Anschließend werden durch Model Checking Safetyeigenschaften wie Schnittstellenkorrektheit und Livenesseigenschaften wie garantierter Fortschritt oder garantierte Freigabe gesperrter Schnittstellen überprüft.

3.2.3 Anforderungen an eine Spezifikations- und Verifikationstechnik für Hochleistungsprotokolle

Bei näherer Betrachtung der oben beschriebenen Spezifikationen und Analysen fällt auf, daß in keiner dieser Arbeiten versucht wurde, formal zu beweisen, daß eine detaillierte, implementierungsnahe Spezifikation eine korrekte Implementierung einer abstrakteren Anforderungsspezifikation ist. Eine explizite, formal beschriebene Anforderungsspezifikation hat mit Ausnahme von Barnard und Crosby (und La Porta und Schwarz, wenn man die abstrakte Protokollbeschreibung von MSP als Anforderungsspezifikation auffaßt) kein Autor entworfen. Auch wurden mit Ausnahme der Arbeit von Barnard und Crosby nur konstruktive Notationen verwendet. Weiterhin fällt auf, daß jede Arbeit ein völlig abgeschlossenes Projekt ist, das nicht die Ergebnisse anderer Arbeiten berücksichtigt. Die durchaus ähnlich gearteten Protokollstrukturen (insbesondere die Protokollfunktionen, siehe Abschnitt 2.2.2 und 2.2.3) wurden also in jedem Projekt neu spezifiziert. Schließlich ist evident, daß die Spezifikationen mit Ausnahme von MSP und PATROCLOS nicht von den Protokollentwicklern, sondern von Personen erstellt wurden, die dem Forschungsbereich der formalen Methoden zugeordnet werden dürfen. Diese Tatsachen sind für uns weitere Indizien dafür, daß formal unterstützte Systementwicklung häufig als zu kompliziert und aufwendig aufgefaßt wird.

In Abschnitt 2.5 haben wir vorgeschlagen, eine Bibliothek von Spezifikationen von Protokollfunktionen und Diensteigenschaften zu entwickeln, aus denen sich Protokollentwickler ihre entsprechenden Protokoll- und Dienstspezifikationen zusammensetzen können. Die formale Verifikation soll durch bereits bewiesene Theoreme unterstützt werden, so daß Protokollentwickler Verfeinerungsbeweise durch einfaches Zusammenfügen von Theoremen zu Implikationsketten führen können. An dieser Stelle wollen wir einige Argumente dafür anführen,

daß sich Leslie Lamports TLA [Lam94] gut als Spezifikationstechnik für diese Bibliothek eignet. Wie schon in Abschnitt 3.1.3 angesprochen wurde, unterstützt TLA sowohl konstruktive als auch deskriptive Systembeschreibungen. TLA erlaubt die Definition von Variablen und ermöglicht die Beschreibung von Zustandsübergängen als transitionsartige Aktionen, so daß ein pseudo-programmiersprachlicher Spezifikationsstil verfolgt werden kann. TLA hat gegenüber den auf erweiterten endlichen Automaten basierenden Methoden wie SDL oder ESTELLE den Vorteil, daß man Liveness explizit definieren kann. Liveness wird in der kanonischen TLA-Beschreibungsform nur durch Fairnessannahmen an Aktionen modelliert, deren Semantik gut verständlich ist (siehe Abschnitt 3.3.2). Aufgrund dieser einfachen Livenessmodellierung können temporallogische Beweise unter Zuhilfenahme einer geringen Anzahl an logischen Deduktionsregeln in eine Kette von Prädikatenlogikbeweisen überführt werden. TLA-Verifikationen sind nach unserer Ansicht wesentlich einfacher als zum Beispiel auf Protokollalgebren basierende Beweise.

Allerdings hat TLA auch einige Nachteile. Es erlaubt die Spezifikation einer großen Menge implementierbarer und auch nicht implementierbarer Spezifikationen. Lamport schlägt mit TLA$^+$ eine Spezifikationssprache vor, die zwar sehr flexible Systemspezifikationen zuläßt, dem Entwickler jedoch keine syntaktischen Konstrukte anbietet, die übersichtliche Spezifikationen komplexer Systeme ermöglichen. Mit cTLA$^+$ wurde deshalb eine Spezifikationssprache entwickelt, die einen prozeßorientierten Beschreibungsstil erlaubt (siehe Abschnitt 4.3). Ein weiteres Manko des reinen TLAs ist die von Abadi und Lamport propagierte Kompositionsmethode (vgl. [AL93a]), die geteilte Variablen und damit asynchrone Prozeßkopplungen vorsieht. In Abschnitt 3.1.4 führten wir jedoch aus, daß sich synchrone Kopplungen für die constraintorientierte Modellierung von Systemen besser eignen als asynchrone Kopplungen. Wir haben deshalb einen kompositionalen Spezifikationsstil entwickelt, der synchrone Prozeßkopplungen verwendet (siehe Abschnitt 4).

3.3 TLA

Die Temporale Logik der Aktionen (Temporal Logic of Actions, TLA) wird von Leslie Lamport in [Lam94] beschrieben. Wir werden zunächst die Symbole von TLA und ihre semantische Bedeutung vorstellen. Anschließend skizzieren wir, wie Systeme in TLA spezifiziert und verifiziert werden. Dabei wird insbesondere auf das Konzept der Refinement Mappings eingegangen. Zum Abschluß stellen wir den Kompositionsansatz von Abadi und Lamport [AL93a] vor.

3.3.1 Syntax und Semantik

Ein Algorithmus wird von Lamport als Zustandstransitionssystem aufgefaßt, in dem Zustände durch Wertbelegungen von Variablen beschrieben werden (vgl. auch [Lam89]). Abläufe von Algorithmen werden durch unendlich lange Folgen von Zuständen modelliert. Die grundlegenden Konstrukte in TLA sind Werte, Variablen, Zustände und Zustandsfolgen. Die Menge Var enthält alle Bezeichner, mit denen man Variablen in TLA-Formeln referenzieren kann. Val ist die Menge aller Werte, die Variablen annehmen dürfen, jedoch nicht die Boolschen Wahrheitswerte *True* und *False*. Da in allen denkbaren TLA-Formeln nur Bezeichner und Werte aus Var und Val verwendet werden, sind keine expliziten Mechanismen zur Deklaration von Variablen und Datentypen notwendig.

Die Semantik in TLA wird durch Zustände und Zustandsfolgen definiert. Ein Zustand s entspricht einer beliebigen Belegung aller Variablen, deren Bezeichner in Var enthalten sind, mit Werten aus Val. St ist die Menge aller möglichen Zustände s. Ein Ablauf eines Algorithmus entspricht einer unendlich langen Zustandsfolge:

$$\langle s_0, s_1, s_2, \ldots \rangle \tag{3.1}$$

Ein Ablauf eines terminierenden Algorithmus wird durch eine Zustandsfolge

$$\langle s_0, s_1, s_2, \ldots, s_n, s_n, s_n, \ldots \rangle \tag{3.2}$$

beschrieben, in der der Zustand s_n, der den Eintritt der Terminierung beschreibt, für immer beibehalten wird. Die Menge aller unendlich langen Zustandsfolgen wird als St^∞ bezeichnet.

Außer den Variablen aus Var, deren Wertbelegungen sich im Ablauf eines Algorithmus ändern können, existieren in TLA auch Konstanten mit festem bekannten oder unbekannten Wert. Eine typische Konstante mit bekanntem Wert ist zum Beispiel die Zahl 2 (die in den Peano-Axiomen für die natürlichen Zahlen den Wert $succ(succ(0))$ bezeichnet). Die Konstante n in $\exists n : n \in Nat$ bezeichnet hingegen einen festen, aber unbekannten Wert[2].

Im folgenden werden wir zunächst nur die Modellierung einzelner Zustände in TLA erläutern. Wir beschreiben die Wertbelegung einer Variablen mit Bezeichner x in einem Zustand s durch die Abbildung $s[\![x]\!]$. Die semantische Bedeutung $s[\![F]\!]$ einer Zustandsfunktion F — eines Terms, der Variablenbezeichner aus Var enthält — im Zustand s ergibt sich dadurch, daß jeder Variablenbezeichner x aus Var durch den Wert $s[\![x]\!]$ ersetzt wird[3]:

$$s[\![F]\!] \triangleq F(\forall v \in \mathsf{Var} : s[\![v]\!]/v) \tag{3.3}$$

Zum Beispiel wird die Wertbelegung der Formel $[\![x^2 + 2y + n - 2]\!]$ im Zustand s durch $(s[\![x]\!])^2 + 2 \cdot (s[\![y]\!]) + n - 2$ beschrieben, falls x und y Variablenbezeichner und 2 bzw. n Konstantenbezeichner sind. Die Operatoren $+$ und $-$ müssen durch einen Satz von Axiomen definiert werden. Die semantische Bedeutung eines Zustandsprädikats, einer Booleschen Formel mit Variablenbezeichnern, wird ebenfalls durch die in Formel 3.3 beschriebene Ersetzungsregel definiert. $[\![\exists n : x^2 < y + n]\!]$ entspricht im Zustand s demnach $\exists n : (s[\![x]\!])^2 < s[\![y]\!] + n$.

Aktionen sind spezielle Prädikate, die sich auf Zustandspaare beziehen. Sie erlauben die Modellierung von Zustandsübergangsrelationen. Zur Unterscheidung werden Variablen im „alten“ Zustand durch einen Bezeichner aus Var und im „neuen“ Zustand durch einen Bezeichner aus Var und dem angehängten Symbol $'$ (Prime) beschrieben. Die Semantik einer Aktion π für einen „alten“ Zustand s und einen „neuen“ Zustand t wird durch

$$s[\![\pi]\!]t \triangleq \pi(\forall v \in \mathsf{Var} : s[\![v]\!]/v, t[\![v]\!]/v') \tag{3.4}$$

definiert. Variablenbezeichner x werden durch den Wert $s[\![x]\!]$ im Zustand s und Variablenbezeichner x' durch den Wert $t[\![x]\!]$ im Zustand t ersetzt. $[\![x^2 < y' + 2]\!]$ entspricht also $(s[\![x]\!])^2 < t[\![y]\!] + 2$.

[2] Aus mathematischer Sicht sind Konstanten mit unbekanntem Wert Variablen und werden von Lamport deshalb als „starre Variablen“ (Rigid Variables) bezeichnet. Um Verwechslungen mit echten Variablen (Flexible Variables) auszuschließen, übernehmen wir diese Bezeichnung nicht.

[3] $\triangleq$ bedeutet „Gleich per Definition“; $F(\forall x, y : x/y)$ bedeutet, daß jeder Bezeichner y in Formel F durch den Bezeichner x ersetzt wird.

Prädikate F können als Aktionen aufgefaßt werden, die Aussagen nur über den „alten“ Zustand machen. Mit $'$ markierte Prädikate

$$F' = F(\forall v \in \mathsf{Var} : v'/v) \tag{3.5}$$

beziehen sich hingegen nur auf den „neuen“ Zustand:

$$\forall s, t \in \mathsf{St} : s[\![F']\!]t = t[\![F]\!] \tag{3.6}$$

Die Schaltbedingungen von Zustandsübergangsrelationen werden durch ein spezielles Prädikat *Enabled* π beschrieben. *Enabled* π gilt in jedem Zustand s, wenn es für s einen Nachfolgezustand t gibt, der durch Schalten von π erreicht wird:

$$s[\![\mathit{Enabled}\ \pi]\!] \triangleq \exists t \in \mathsf{St} : s[\![\pi]\!]t \tag{3.7}$$

Die Semantik einzelner Zustände dient als Grundlage zur Beschreibung der Semantik von Zustandsfolgen (siehe Formel 3.1), auf die wir im folgenden eingehen. Die semantische Bedeutung einer TLA-Formel T in Bezug auf eine Zustandsfolge $\langle s_0, s_1, s_2, \ldots\rangle$ beschreiben wir durch $\langle s_0, s_1, s_2, \ldots\rangle[\![T]\!]$. Die Abhängigkeit zwischen Zuständen und Zustandsfolgen ist dadurch gegeben, daß ein Prädikat F sich immer nur auf den ersten Zustand einer Zustandsfolge bezieht, während eine Aktion π Aussagen über die ersten beiden Zustände einer Zustandsfolge macht:

$$\langle s_0, s_1, s_2, \ldots\rangle[\![F]\!] \triangleq s_0[\![F]\!] \tag{3.8}$$

$$\langle s_0, s_1, s_2, \ldots\rangle[\![\pi]\!] \triangleq s_0[\![\pi]\!]s_1 \tag{3.9}$$

Um aber auch das Verhalten aller anderen Zustände von Zustandsfolgen treffen zu können, führt Lamport eine einfache temporale Linearzeitlogik ein. Sie besteht aus den Operatoren der Aussagenlogik, dem in den Aktionen verwendeten Symbol $'$ (Prime), sowie den zwei temporalen Operatoren $\Box$ und $\exists$. Von $\Box$ werden zusätzlich die temporalen Operatoren $\Diamond$ und $\rightsquigarrow$ abgeleitet. Da alle propositionalen Operatoren aus den beiden Operatoren $\wedge$ und $\neg$ hergeleitet werden können, geben wir hier nur die semantische Bedeutung dieser Operatoren für Zustandsfolgen an, wobei T und U Temporalformeln beschreiben:

$$\langle s_0, s_1, s_2, \ldots\rangle[\![T \wedge U]\!] \triangleq \langle s_0, s_1, s_2, \ldots\rangle[\![T]\!] \wedge \langle s_0, s_1, s_2, \ldots\rangle[\![U]\!] \tag{3.10}$$

$$\langle s_0, s_1, s_2, \ldots\rangle[\![\neg T]\!] \triangleq \neg\langle s_0, s_1, s_2, \ldots\rangle[\![T]\!] \tag{3.11}$$

Der temporale Operator $\Box$ („immer“, „always“) wird durch

$$\langle s_0, s_1, s_2, \ldots\rangle[\![\Box T]\!] \triangleq \forall n \in \mathit{Nat} : \langle s_n, s_{n+1}, s_{n+2}, \ldots\rangle[\![T]\!] \tag{3.12}$$

definiert. Aus dieser Definition folgt, daß die Formel T in allen Zuständen s_n der Zustandsfolge gilt. Somit kann man die semantische Bedeutung von Prädikaten F und Aktionen π durch

$$\begin{aligned} \langle s_0, s_1, s_2, \ldots\rangle[\![\Box F]\!] &\equiv \forall n \in \mathit{Nat} : s_n[\![F]\!] \\ \langle s_0, s_1, s_2, \ldots\rangle[\![\Box \pi]\!] &\equiv \forall n \in \mathit{Nat} : s_n[\![\pi]\!]s_{n+1} \end{aligned} \tag{3.13}$$

beschreiben (siehe Formeln 3.8 und 3.9).

Von $\Box$ werden die Operatoren $\Diamond$ („irgendwann“, „eventually“) und $\leadsto$ („führt zu“, „leads to“) abgeleitet:

$$\Diamond T \quad \triangleq \quad \neg\Box\neg T \tag{3.14}$$

$$T \leadsto U \quad \triangleq \quad \Box(T \Rightarrow \Diamond U) \tag{3.15}$$

Durch $\Diamond T$ wird beschrieben, daß es in einer Zustandsfolge mindestens einen Zustand s_n gibt, in dem T gültig ist (T ist irgendwann im Ablauf des Algorithmus wahr):

$$\langle s_0, s_1, s_2, \ldots\rangle[\![\Diamond T]\!] \equiv \exists n \in Nat : s_n[\![T]\!] \tag{3.16}$$

Der binäre Operator $T \leadsto U$ sagt aus, daß immer, wenn in irgendeinem Zustand s_n die Temporalformel T wahr ist, im gleichen oder einem „späteren“ Zustand s_{n+m} auch die Temporalformel U gültig ist (T führt also zu U). Aus $\Box$ und $\Diamond$ kann man die Operationen $\Box\Diamond T$ und $\Diamond\Box T$ zusammensetzen. $\Box\Diamond T$ sagt aus, daß T in unendlich vielen Zuständen der Zustandsfolge gilt. $\Diamond\Box T$ bedeutet, daß T ab einem bestimmten Zustand s_n in allen Zuständen s_{n+m} wahr ist. Es gibt also nur endlich viele Zustände, in denen T ungültig ist.

Durch den Existenzquantor $\exists$ („Bold Exists“) kann man einzelne Zustandskomponenten ausblenden.

$$U \equiv \exists w : T \tag{3.17}$$

bedeutet, daß sich das durch U beschreibene Verhalten von dem durch T modellierten Verhalten darin unterscheidet, daß die Werte der Variable w in U nicht berücksichtigt werden. Bevor wir die Semantik von $\exists$ angeben, führen wir zunächst einige Hilfskonstrukte ein[4]:

$$s =_w t \quad \triangleq \quad \forall v \in \mathsf{Var} \setminus \{w\} : s[\![v]\!] = t[\![v]\!] \tag{3.18}$$

$$\sigma =_w \tau \quad \triangleq \quad \forall v \in \mathsf{Var} \setminus \{w\} : \sigma[\![v]\!] = \tau[\![v]\!] \tag{3.19}$$

Zwei Zustände (s, t) oder Zustandsfolgen (σ, τ), sind „gleich mit Ausnahme von w“ $(=_w)$, wenn alle Variablen außer w den gleichen Wert oder die gleiche Folge von Werten haben. Außerdem führen wir einen Operator $\natural$ ein, der Stotterschritte in Zustandsfolgen ausschließt. Unter einem Stotterschritt verstehen wir einen Zustandsübergang, in dem die beiden benachbarten Zustände den gleichen Wert haben. Durch den Operator $\natural$ wird eine Zustandsfolge

$$\sigma \triangleq \langle s_0, s_1, s_2, \ldots, s_n, s_n, s_n, \ldots, s_n, s_{n+1}, \ldots\rangle \tag{3.20}$$

auf eine Zustandsfolge abgebildet, in der alle Stotterschritte entfernt sind:

$$\natural\sigma \triangleq \langle s_0, s_1, s_2, \ldots, s_n, s_{n+1}, \ldots\rangle \text{ mit } \forall i : s_i \neq s_{i+1} \tag{3.21}$$

Man definiert nun die Semantik von $\exists$ durch

$$\sigma[\![\exists w : T]\!] \triangleq \exists \rho, \tau \in \mathsf{St}^\infty : (\natural\sigma = \natural\rho) \wedge (\rho =_w \tau) \wedge \tau[\![T]\!] \tag{3.22}$$

Eine Zustandsfolge σ erfüllt also $\exists w : T$, wenn T in einer anderen Zustandsfolge τ gültig ist, die sich von σ nur durch Stotterschritte sowie durch die Werte der Variablen w unterscheidet. Die tatsächliche Wertbelegung von w ist also uninteressant, solange es mindestens eine gültige Wertbelegung gibt. Wie der normale Existenzquantor in der Prädikatenlogik ist $\exists$ distributiv gegenüber der Disjunktion:

$$(\exists w : T \vee U) \equiv (\exists w : T) \vee (\exists w : U) \tag{3.23}$$

Syntax

$\langle formula \rangle \triangleq \langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg\langle formula \rangle$
$\mid \langle formula \rangle \wedge \langle formula \rangle \mid \Box\, \langle formula \rangle$

$\langle action \rangle \triangleq$ boolean-valued expression containing constant symbols, variables, and primed variables

$\langle predicate \rangle \triangleq \langle action \rangle$ with no primed variables $\mid Enabled\langle action \rangle$

$\langle state\ function \rangle \triangleq$ nonboolean expression containing constant symbols and variables

Semantics

$s[[F]] \triangleq F(\forall\, v \in \mathsf{Var} : s[[v]]/v)$

$s[[\pi]]t \triangleq \pi(\forall\, v \in \mathsf{Var} : s[[v]]/v,\ t[[v]]/v')$

$\models \pi \triangleq \forall s, t \in \mathsf{St} : s[[\pi]]t$

$\sigma[[T \wedge U]] \triangleq \sigma[[T]] \wedge \sigma[[U]]$

$\sigma[[\neg T]] \triangleq \neg\sigma[[T]]$

$\models F \triangleq \forall \sigma \in \mathsf{St}^\infty : \sigma[[F]]$

$s[[Enabled\ \pi]] \triangleq \exists t \in \mathsf{St} : s[[\pi]]t$

$\langle s_0, s_1, \ldots\rangle[[\Box T]] \triangleq \forall n \in Nat : \langle s_n, s_{n+1}, \ldots\rangle[[T]]$

$\langle s_0, s_1, \ldots\rangle[[\pi]] \triangleq s_0[[\pi]]s_1$

Additional notation

$F' \triangleq F(\forall\, v \in \mathsf{Var} : v'/v)$

$[\pi]_w \triangleq \pi \vee (w' = w)$

$\langle\pi\rangle_w \triangleq \pi \wedge (w' \neq w)$

$Unchanged\ w \triangleq w' = w$

$\Diamond T \triangleq \neg\Box\neg T$

$T \rightsquigarrow U \triangleq \Box(T \Rightarrow \Diamond U)$

$WF_w(\pi) \triangleq \Box\Diamond\ \langle\pi\rangle_w \vee \Box\Diamond\neg Enabled\langle\pi\rangle_w$

$SF_w(\pi) \triangleq \Box\Diamond\ \langle\pi\rangle_w \vee \Diamond\Box\neg Enabled\langle\pi\rangle_w$

where F is a $\langle state\ function \rangle$; π is an $\langle action \rangle$ or $\langle predicate \rangle$; T and U are $\langle formula \rangle$ s; w is a $\langle state\ function \rangle$

s, s_0, s_1, ... are states; σ is a behavior; $(\forall\, v \in \mathsf{Var} : \ldots/v, \ldots/v')$ denotes substitution for all variables v

Abbildung 3.1: TLA (entnommen aus [Lam94] mit geänderten Bezeichnern)

Die grundlegenden Konstrukte der Syntax und Semantik von TLA werden in der Abbildung 3.1 zusammengefaßt. Abbildung 3.2 enthält einen von Lamport aufgestellten Satz an Axiomen und Deduktionsregeln, der logisches Schließen auf TLA-Formeln erlaubt. Die Definition und Beweisregeln für $\exists$ und einen weiteren Existenzquantor $\exists$ für Konstanten ist in Abbildung 3.3 aufgeführt.

Die Allgemeingültigkeit $\models$ der Prädikate F, Aktionen π und Temporalformeln T wird durch

$$\begin{aligned} \models F &\triangleq \forall s \in \mathsf{St} : s[\![F]\!] \\ \models \pi &\triangleq \forall s, t \in \mathsf{St} : s[\![\pi]\!]t \\ \models T &\triangleq \forall \sigma \in \mathsf{St}^\infty : \sigma[\![T]\!] \end{aligned} \tag{3.24}$$

definiert. Lamport hat bewiesen, daß F, π und T nur durch die in den Abbildungen 3.2 und 3.3 aufgeführten Axiome und Regeln deduziert werden können ($\vdash F$, $\vdash \pi$, $\vdash T$), wenn sie

[4] $s, t \in \mathsf{St}$; $\sigma, \tau \in \mathsf{St}^\infty$

The Rules of Simple Temporal Logic

STL1. $\dfrac{T \text{ provable by propositional logic}}{\Box T}$

STL2. $\vdash \Box T \;\Rightarrow\; T$

STL3. $\vdash \Box\Box T \;\equiv\; \Box T$

STL4. $\dfrac{T \Rightarrow U}{\Box T \;\Rightarrow\; \Box U}$

STL5. $\vdash \Box(T \wedge U) \;\equiv\; (\Box T) \wedge (\Box U)$

STL6. $\vdash (\Diamond\Box T) \wedge (\Diamond\Box U) \;\equiv\; \Diamond\Box(T \wedge U)$

STL7. $\vdash \Box\Diamond\Box T \;\equiv\; \Diamond\Box T$

LATTICE. $\succ$ a well-founded partial order on a set S

$$\frac{T \wedge (\mathsf{c} \in \mathsf{S}) \;\Rightarrow\; (H_{\mathsf{c}} \rightsquigarrow (U \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{T \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \rightsquigarrow U)}$$

The Basic Rules of TLA

TLA1. $\dfrac{F \wedge (w' = w) \;\Rightarrow\; F'}{\Box F \;\equiv\; F \wedge \Box[F \Rightarrow F']_w}$

TLA2. $\dfrac{F \wedge [\alpha]_w \;\Rightarrow\; G \wedge [\beta]_u}{\Box F \wedge \Box[\alpha]_w \;\Rightarrow\; \Box G \wedge \Box[\beta]_u}$

Additional Rules

INV1. $\dfrac{I \wedge [\pi]_w \Rightarrow I'}{I \wedge \Box[\pi]_w \Rightarrow \Box I}$

INV2. $\vdash \Box I \;\Rightarrow\; (\Box[\pi]_w \;\equiv\; \Box[\pi \wedge I \wedge I']_w)$

WF1.

$$\frac{\begin{array}{l} F \wedge [\pi]_w \;\Rightarrow\; (F' \vee G') \\ F \wedge \langle \pi \wedge \alpha \rangle_w \;\Rightarrow\; G' \\ F \;\Rightarrow\; \mathit{Enabled}\ \langle \alpha \rangle_w \end{array}}{\Box[\pi]_w \wedge \mathrm{WF}_w(\alpha) \;\Rightarrow\; (F \rightsquigarrow G)}$$

WF2.

$$\frac{\begin{array}{l} \langle \pi \wedge \beta \rangle_w \;\Rightarrow\; \langle \overline{\lambda} \rangle_{\overline{u}} \\ F \wedge F' \wedge \langle \pi \wedge \alpha \rangle_w \wedge \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \;\Rightarrow\; \beta \\ F \wedge \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \;\Rightarrow\; \mathit{Enabled}\ \langle \alpha \rangle_w \\ \Box[\pi \wedge \neg\beta]_w \wedge \mathrm{WF}_w(\alpha) \wedge \Box T \\ \qquad \wedge \Diamond\Box \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \;\Rightarrow\; \Diamond\Box F \end{array}}{\Box[\pi]_w \wedge \mathrm{WF}_w(\alpha) \wedge \Box T \;\Rightarrow\; \overline{\mathrm{WF}_u(\lambda)}}$$

SF1.

$$\frac{\begin{array}{l} F \wedge [\pi]_w \;\Rightarrow\; (F' \vee G') \\ F \wedge \langle \pi \wedge \alpha \rangle_w \;\Rightarrow\; G' \\ \Box F \wedge \Box[\pi]_w \wedge \Box T \;\Rightarrow\; \Diamond \mathit{Enabled}\ \langle \alpha \rangle_w \end{array}}{\Box[\pi]_w \wedge \mathrm{SF}_w(\alpha) \wedge \Box T \;\Rightarrow\; (F \rightsquigarrow G)}$$

SF2.

$$\frac{\begin{array}{l} \langle \pi \wedge \beta \rangle_w \;\Rightarrow\; \langle \overline{\lambda} \rangle_{\overline{u}} \\ F \wedge F' \wedge \langle \pi \wedge \alpha \rangle_w \;\Rightarrow\; \beta \\ F \wedge \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \;\Rightarrow\; \mathit{Enabled}\ \langle \alpha \rangle_w \\ \Box[\pi \wedge \neg\beta]_w \wedge \mathrm{SF}_w(\alpha) \wedge \Box T \\ \qquad \wedge \Box\Diamond \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \;\Rightarrow\; \Diamond\Box F \end{array}}{\Box[\pi]_w \wedge \mathrm{SF}_w(\alpha) \wedge \Box T \;\Rightarrow\; \overline{\mathrm{SF}_u(\lambda)}}$$

where T, U, H_{c} are TLA formulas — F, G, I are predicates
α, β, π, λ are actions — w, u are state functions

Abbildung 3.2: Axiome und Beweisregeln von TLA (entnommen aus [Lam94] mit geänderten Bezeichnern)

allgemeingültig sind:

$$\begin{array}{lcl} \vdash F & \Rightarrow & \models F \\ \vdash \pi & \Rightarrow & \models \pi \\ \vdash T & \Rightarrow & \models T \end{array} \tag{3.25}$$

Syntax

$\langle general\ formula \rangle \triangleq \langle formula \rangle \mid \exists \langle variable \rangle : \langle general\ formula \rangle$
$\mid \exists \langle rigid\ variable \rangle : \langle general\ formula \rangle$
$\mid \langle general\ formula \rangle \wedge \langle general\ formula \rangle$
$\mid \neg \langle general\ formula \rangle$

$\langle formula \rangle \triangleq$ a simple TLA formula (siehe Abbildung 3.1)

Semantics

$\langle s_0, s_1, \ldots \rangle =_w \langle t_0, t_1, \ldots \rangle \triangleq \forall\, \mathsf{n} \in \mathsf{Nat} : \forall\, v \in \mathsf{Var} \setminus \{w\} : s_{\mathsf{n}}[\![v]\!] = t_{\mathsf{n}}[\![v]\!]$

$\natural\langle s_0, s_1, s_2, \ldots \rangle \triangleq$ **if** $\forall \mathsf{n} \in \mathsf{Nat} : s_{\mathsf{n}} = s_0$
then $\langle s_0, s_0, s_0, \ldots \rangle$
else if $s_1 = s_0$ **then** $\natural\langle s_1, s_2, s_3, \ldots \rangle$
else $\langle s_0 \rangle \circ \natural\langle s_1, s_2, \ldots \rangle$

$\sigma[\![\exists\, w : T]\!] \triangleq \exists\, \rho, \tau \in \mathbf{St}^\infty : (\natural\sigma = \natural\rho) \wedge (\rho =_w \tau) \wedge \tau[\![T]\!]$

$\sigma[\![\exists\, \mathsf{c} : T]\!] \triangleq \exists \mathsf{c} \in \mathsf{Val} : \sigma[\![T]\!]$

Proof Rules

$E1.\ \vdash T(f/w) \Rightarrow \exists\, w : T$

$E2.\ \dfrac{T \Rightarrow U \quad w \text{ does not occur free in } U}{(\exists\, w : T) \Rightarrow U}$

$F1.\ \vdash T(\mathsf{e}/\mathsf{c}) \Rightarrow \exists\, \mathsf{c} : T$

$F2.\ \dfrac{T \Rightarrow U \quad \mathsf{c} \text{ does not occur free in } U}{(\exists\, \mathsf{c} : T) \Rightarrow U}$

where w is a $\langle variable \rangle$ — T, U are $\langle general\ formula \rangle$s
f is a state function — s, s_0, t_0, s_1, t_1, ... are states
c is a $\langle rigid\ variable \rangle$ — σ is a behavior
e is a constant expression — $\circ$ denotes concatenation of sequences

Abbildung 3.3: Quantoren in TLA (entnommen aus [Lam94] mit geänderten Bezeichnern)

3.3.2 Spezifikation

Zustandsfolgen, deren Zustände durch Wertbelegungen von Variablen dargestellt werden, eignen sich hervorragend zur Verhaltensbeschreibung von aus gekoppelten von-Neumann-Rechnern bestehenden nebenläufigen Systemen, da jeder zulässige Ablauf eines derartigen Systems durch eine Zustandsfolge modelliert wird. Die Menge Π der Zustandsfolgen aller zulässigen Abläufe des Systems beschreibt das von ihm geforderte Verhalten ($\Pi \subseteq \mathsf{St}^\infty$). Das System kann durch eine Temporalformel Ψ modelliert werden, die für genau alle Zustandsfolgen aus Π wahr ist:

$$\forall \sigma \in \mathsf{St}^\infty : \sigma[\![\Psi]\!] \Leftrightarrow \sigma \in \Pi \tag{3.26}$$

Lamport führt zur Systembeschreibung einen kanonischen TLA-Stil für Temporalformeln ein, den wir im folgenden vorstellen. Dieser Stil erlaubt zum einen übersichtliche Beschreibungen. Zum anderen bildet er die Basis für einfache und schematische Systemverifikationen. Die Beschreibungen von Safety- und Livenesseigenschaften eines Systems (siehe Abschnitt 3.1.1)

werden im kanonischen Stil voneinander strikt getrennt. Safetyeigenschaften werden durch Zustandstransitionssysteme modelliert, die durch eine Menge I von Anfangszuständen und eine Menge Δ der erlaubten Transitionen definiert sind. Die Menge Π_s aller Zustandsfolgen des zu spezifizierenden Systems, die die geforderten Safetyeigenschaften einhalten, hängt von I und Δ ab:

$$\Pi_s \triangleq \{\langle s_0, s_1, s_2, \ldots\rangle | s_0 \in I\} \cap \{\langle s_0, s_1, s_2, \ldots\rangle | \forall n \in Nat : (s_n, s_{n+1}) \in \Delta \vee s_n = s_{n+1}\} \quad (3.27)$$

Nach Formel 3.8 kann man I durch ein Prädikat $Init$ beschreiben:

$$\forall s \in \mathsf{St} : s[\![Init]\!] \Leftrightarrow s \in I \quad (3.28)$$

Die Menge Δ der erlaubten Transitionen wird durch eine Aktion π beschrieben (siehe Formel 3.9):

$$\forall s, t \in \mathsf{St} : s[\![\pi]\!]t \Leftrightarrow (s, t) \in \Delta \quad (3.29)$$

Außer den Zustandsübergängen aus Δ werden auch Stotterschritte toleriert. Dadurch sind Beweise möglich, daß das durch die Formel Ψ modellierte System von einem detaillierteren System erbracht wird[5]. Die Menge aller in Π_s erlaubten Zustandsübergänge wird somit durch die Formel $\pi \vee (w' = w)$ beschrieben, wobei w das Tupel aller Bezeichner der in einer Spezifikation eingesetzten Variablen ist. Lamport führt dafür die vereinfachte Schreibweise

$$[\pi]_w \triangleq \pi \vee Unchanged\ w \quad (3.30)$$

ein. $Unchanged\ w$ steht dabei für $(w' = w)$. $\Box[\pi]_w$ wird als Zustandsfolgerelation bzw. Zustandsübergangsrelation (Next-Step-Relation)

des durch Ψ modellierten Systems bezeichnet. Die Zustandsfolgen aus Π_s werden durch die TLA-Formel $Init \wedge \Box[\pi]_w$ beschrieben:

$$\forall \sigma \in \mathsf{St}^\infty : \sigma[\![Init \wedge \Box[\pi]_w]\!] \Leftrightarrow \sigma \in \Pi_s \quad (3.31)$$

Als Beispiel spezifizieren wir einen sehr einfachen Semaphor (vgl. [Dij68]), dessen Safetyeigenschaften in der Formel Ψ_s modelliert sind:

$$\begin{aligned}
Init_\Psi &\triangleq x = \text{'leer'} \\
P_{1_\Psi} &\triangleq x = \text{'leer'} \wedge x' = 1 \\
P_{2_\Psi} &\triangleq x = \text{'leer'} \wedge x' = 2 \\
V_\Psi &\triangleq x' = \text{'leer'} \\
\pi_\Psi &\triangleq P_{1_\Psi} \vee P_{2_\Psi} \vee V_\Psi \\
\Psi_s &\triangleq Init_\Psi \wedge \Box[\pi_\Psi]_{(x)}
\end{aligned} \quad (3.32)$$

Zwei Prozesse bewerben sich um einen kritischen Abschnitt, in dem sich zu jedem Zeitpunkt immer höchstens ein Prozeß aufhalten darf. Vor dem Eintritt in den kritischen Abschnitt müssen die Prozesse 1 und 2 die Aktionen P_{1_Ψ} bzw. P_{2_Ψ} (Passieren, niederl. Passeren) schalten. Nachdem sie ihn verlassen haben, schalten sie die Aktion V_Ψ (Freigeben, niederl. Vrijgeven). Aufgrund der Schaltbedingung von Aktion P_{2_Ψ} kann Prozeß 2 nicht in den kritischen

[5]Zustandsübergänge im detaillierteren System, in denen nur in Ψ nicht auftretende Variablen verändert werden, kann man auf einen Stotterschritt in Ψ abbilden.

Abschnitt eintreten, wenn Prozeß 1 sich noch darin befindet. Nachdem er in den kritischen Abschnitt eingetreten ist, kann hingegen Prozeß 1 nicht mehr die Aktion P_{1_Ψ} schalten, solange 2 nicht V_Ψ schaltet. Damit ist also ausgeschlossen, daß beide Prozesse sich gleichzeitig im kritischen Abschnitt befinden.

Die Liveness eines Systems wird im kanonischen Stil durch schwache und starke Fairnessanforderungen an Aktionen modelliert (vgl. [Fra86, MP92a]). Die schwache Fairness einer Aktion λ, die eine Teilmenge der Transitionen aus Δ beschreibt ($\lambda \Rightarrow \pi$), wird durch die Formel

$$WF_w(\lambda) \triangleq \Box\Diamond\neg Enabled\ \langle\lambda\rangle_w \vee \Box\Diamond\langle\lambda\rangle_w \tag{3.33}$$

definiert, wobei $\langle\lambda\rangle_w$ den Ausschluß von Stotterschritten ausdrückt:

$$\langle\lambda\rangle_w \triangleq \lambda \wedge \neg Unchanged\ w \tag{3.34}$$

Die schwache Fairness erzwingt Zustandsfolgen, in denen unendlich viele Zustandsübergänge einer Ausführung der Aktion $\langle\lambda\rangle_w$ entsprechen oder in denen die Aktion λ in unendlich vielen Zuständen nicht schaltbar ist.

Die starke Fairness einer Aktion λ wird durch die Formel

$$SF_w(\lambda) \triangleq \Diamond\Box\neg Enabled\ \langle\lambda\rangle_w \vee \Box\Diamond\langle\lambda\rangle_w \tag{3.35}$$

definiert. Dadurch werden Zustandsfolgen erzwungen, in denen unendlich viele Zustandsübergänge einer Ausführung der Aktion $\langle\lambda\rangle_w$ entsprechen oder in denen die Aktion $\langle\lambda\rangle_w$ nur in endlich vielen Zuständen schaltbar ist. Mit Hilfe der starken Fairness kann man die faire Ausführung mehrerer miteinander konkurrierender Aktionen sicherstellen.

Das in Formel 3.32 aufgeführte Semaphorbeispiel wird nun um Fairnessanforderungen erweitert. Das erweiterte System wird durch die Formel Ψ beschrieben:

$$\Psi \triangleq Init_\Psi \wedge \Box[\pi_\Psi]_{(x)} \wedge SF_{(x)}(P_{1_\Psi}) \wedge SF_{(x)}(P_{2_\Psi}) \wedge WF_{(x)}(V_\Psi) \tag{3.36}$$

Durch die schwache Fairness $WF_{(x)}(V_\Psi)$ wird sichergestellt, daß kein Prozeß ewig im kritischen Abschnitt verharrt. Beim Eintritt des Prozesses in den Abschnitt wird die Aktion V_Ψ schaltbar. Sie bleibt solange schaltbar, bis sie geschaltet wird. Um der Fairnessanforderung zu genügen, wird V_Ψ irgendwann geschaltet. Dagegen reicht die schwache Fairness nicht aus, um sicherzustellen, daß beide Prozesse beim Eintritt in den kritischen Abschnitt fair behandelt werden. Eine Verhaltensfolge, in der zwar Prozeß 1 aber niemals Prozeß 2 in den kritischen Abschnitt eintritt, erfüllt die schwache Fairness von P_{1_Ψ} und P_{2_Ψ}, da P_{2_Ψ} immer nicht geschaltet werden darf, wenn Prozeß 1 sich gerade im kritischen Abschnitt befindet. P_{2_Ψ} ist somit unendlich oft nicht schaltbar, wodurch seine schwache Fairness erfüllt wird. Dagegen erzwingt die starke Fairness der Aktionen P_{1_Ψ} und P_{2_Ψ} die Anforderung nach einer fairen Behandlung der beiden Prozesse.

3.3.3 Verifikation

In TLA können außer den Algorithmen auch allgemeine oder spezifische Systemeigenschaften, die von den Algorithmen erfüllt werden sollen, durch Zustandsfolgen beschrieben werden. Ω sei die Menge aller Zustandsfolgen, die eine bestimmte Systemeigenschaft erfüllen. Ein Algorithmus, dessen erlaubte Zustandsfolgen in der Menge Π enthalten sind, erfüllt die gewünschte Systemeigenschaft, wenn

$$\Pi \subseteq \Omega \tag{3.37}$$

gilt. Das kann man auf die Deduktion

$$\vdash T \Rightarrow U \tag{3.38}$$

zurückführen, wenn Π durch die Temporalformel T und Ω durch die Temporalformel U modelliert wird (siehe Deduktionsregeln in Abbildung 3.2).

Zunächst zeigen wir, wie man mit TLA beweisen kann, daß Systeme Systemeigenschaften erfüllen. Dazu stellen wir in Formel Φ eine etwas detailliertere Spezifikation des in Abschnitt 3.3.2 eingeführten Semaphors vor:

$$\begin{array}{rcl}
Init_\Phi & \triangleq & z_1 = \text{'aus'} \wedge z_2 = \text{'aus'} \wedge ka = \text{'fertig'} \\
P_{1_\Phi} & \triangleq & z_1 = \text{'aus'} \wedge z_2 = \text{'aus'} \wedge z_1' = \text{'in'} \wedge ka' = \text{'aktiv'} \wedge Unchanged\ z_2 \\
P_{2_\Phi} & \triangleq & z_1 = \text{'aus'} \wedge z_2 = \text{'aus'} \wedge z_2' = \text{'in'} \wedge ka' = \text{'aktiv'} \wedge Unchanged\ z_1 \\
V_{1_\Phi} & \triangleq & z_1 = \text{'in'} \wedge ka = \text{'fertig'} \wedge z_1' = \text{'aus'} \wedge Unchanged\ (z_2, ka) \\
V_{2_\Phi} & \triangleq & z_2 = \text{'in'} \wedge ka = \text{'fertig'} \wedge z_2' = \text{'aus'} \wedge Unchanged\ (z_1, ka) \\
F_\Phi & \triangleq & ka = \text{'aktiv'} \wedge ka' = \text{'fertig'} \wedge Unchanged\ (z_1, z_2) \\
\pi_\Phi & \triangleq & P_{1_\Phi} \vee P_{2_\Phi} \vee V_{1_\Phi} \vee V_{2_\Phi} \vee F_\Phi \\
w_\Phi & \triangleq & (z_1, z_2, ka) \\
\Phi & \triangleq & Init_\Phi \wedge \Box[\pi_\Phi]_{w_\Phi} \wedge \\
 & & SF_{w_\Phi}(P_{1_\Phi}) \wedge SF_{w_\Phi}(P_{2_\Phi}) \wedge WF_{w_\Phi}(V_{1_\Phi}) \wedge WF_{w_\Phi}(V_{2_\Phi}) \wedge WF_{w_\Phi}(F_\Phi)
\end{array} \tag{3.39}$$

Wir verwenden nun zwei Variablen z_1 und z_2 zur Beschreibung, ob sich Prozeß 1 bzw. 2 im (Wert 'in') oder außerhalb (Wert 'aus') des kritischen Abschnitts befindet. Zur Unterscheidung, welcher Prozeß den kritischen Abschnitt verläßt, zerlegen wir die Aktion V_Ψ in zwei Aktionen V_{1_Φ} und V_{2_Φ}. Außerdem modellieren wir explizit, daß ein Prozeß im kritischen Abschnitt eine Tätigkeit verrichtet und ihn erst nach deren Abschluß verlassen darf. Wir verwenden dazu eine Variable ka, die anzeigt, ob der sich im kritischen Abschnitt befindliche Prozeß gerade aktiv ist (Wert 'aktiv') oder seine Tätigkeit bereits beendet hat (Wert 'fertig'). Der Abschluß der Tätigkeit wird durch eine neue Aktion F_Φ modelliert.

Wenn man die Variable ka im Gegensatz zu z_1 und z_2 nur als Hilfskonstrukt ansieht, dessen Ablauf gegenüber dem externen Betrachter der Spezifikation verdeckt werden soll, kann man sie mit Hilfe des Operators $\exists$ ausblenden:

$$\exists ka : \Phi \tag{3.40}$$

Eine zentrale Eigenschaft des in Formel Φ spezifizierten Semaphors ist, daß sich immer höchstens ein Prozeß im kritischen Abschnitt befindet, was durch die Invariante

$$I \triangleq \Box(z_1 = \text{'aus'} \vee z_2 = \text{'aus'}) \tag{3.41}$$

bestimmt wird. Zu beweisen ist also

$$\Phi \Rightarrow I \tag{3.42}$$

Invariantenbeweise können mit Hilfe der TLA-Regel INV1 (siehe Abbildung 3.2) geführt werden. Man muß dazu nachweisen, daß die initiale Bedingung $Init_\Phi$ die Invariante erfüllt.

Außerdem muß I über alle Aktionen P_{1_Φ}, P_{2_Φ}, V_{1_Φ}, V_{2_Φ} und F_Φ der Zustandsfolgerelation von Φ stabil sein:

$$\begin{array}{lll} Init_\Phi \Rightarrow I & I \wedge P_{1_\Phi} \Rightarrow I' & I \wedge P_{2_\Phi} \Rightarrow I' \\ I \wedge V_{1_\Phi} \Rightarrow I' & I \wedge V_{2_\Phi} \Rightarrow I' & I \wedge F_\Phi \Rightarrow I' \end{array} \tag{3.43}$$

Diese Beweise sind trivial und bleiben dem Leser überlassen. Die Stabilität der Invarianten über alle Aktionen impliziert die Stabilität von π_Φ und aufgrund der Regel INV1 gilt

$$I \wedge \Box[\pi_\Phi]_w \Rightarrow \Box I \tag{3.44}$$

Da I auch von $Init_\Phi$ erfüllt wird, folgt daraus jedoch sofort die Behauptung 3.42.

Da Invarianten immer gültig sind, kann die Zustandsübergangsrelation $\Box[\pi_\Phi]_w$ durch die Relation $\Box[\pi_\Phi \wedge I \wedge I']_w$ ersetzt werden (siehe TLA-Regel INV2 in Abbildung 3.2).

Außer der Überprüfung von Systemeigenschaften sind vor allem Verfeinerungsbeweise interessant (siehe Abschnitt 3.1.1). Wir wollen im weiteren verifizieren, daß der in Φ spezifizierte detaillierte Semaphor eine Verfeinerung des in Ψ beschriebenen Semaphors ist. Der Beweis $\Phi \Rightarrow \Psi$ kann allerdings nicht direkt geführt werden, weil Φ andere Variablen verwendet als Ψ. Wir müssen also die Variablen von Φ auf die Variablen von Ψ abbilden und definieren deshalb eine abgewandelte Spezifikation

$$\overline{\Psi} \triangleq \Psi(\overline{x}/x) \tag{3.45}$$

in der die Variable x durch die Zustandsfunktion $\overline{x}$ ersetzt wird, die über den Variablen z_1 und z_2 definiert ist:

$$\begin{array}{lll} \overline{x} \triangleq \textbf{CASE} & z_1 = \text{'aus'} \wedge z_2 = \text{'aus'} & \rightarrow \text{'leer'} \\ & z_1 = \text{'in'} \wedge z_2 = \text{'aus'} & \rightarrow 1 \\ & z_1 = \text{'aus'} \wedge z_2 = \text{'in'} & \rightarrow 2 \\ & \textbf{OTHERWISE} & \rightarrow \text{'illegal'} \end{array} \tag{3.46}$$

$\overline{\Psi}$ hat dann die Form

$$\begin{array}{rcl} \overline{Init_\Psi} & \triangleq & z_1 = \text{'aus'} \wedge z_2 = \text{'aus'} \\ \overline{P_{1_\Psi}} & \triangleq & z_1 = \text{'aus'} \wedge z_2 = \text{'aus'} \wedge z_1' = \text{'in'} \wedge z_2' = \text{'aus'} \\ \overline{P_{2_\Psi}} & \triangleq & z_1 = \text{'aus'} \wedge z_2 = \text{'aus'} \wedge z_1' = \text{'aus'} \wedge z_2' = \text{'in'} \\ \overline{V_\Psi} & \triangleq & z_1' = \text{'aus'} \wedge z_2' = \text{'aus'} \\ \overline{\pi_\Psi} & \triangleq & \overline{P_{1_\Psi}} \vee \overline{P_{2_\Psi}} \vee \overline{V_\Psi} \\ \overline{\Psi} & \triangleq & \overline{Init_\Psi} \wedge \Box[\overline{\pi_\Psi}]_{(z_1,z_2)} \wedge \\ & & SF_{(z_1,z_2)}(\overline{P_{1_\Psi}}) \wedge SF_{(z_1,z_2)}(\overline{P_{2_\Psi}}) \wedge WF_{(z_1,z_2)}(\overline{V_\Psi}) \end{array} \tag{3.47}$$

Die Spezifikation $\overline{\Psi}$ modelliert kein anderes Verhalten als Ψ. Die Zustände werden anstatt durch die Werte der Variablen x durch die Werte der Variablen z_1 und z_2 repräsentiert. Man kann durch einen Invariantenbeweis verifizieren, daß x nur die Werte 'leer', 1 und 2 annehmen kann, die in $\overline{\Psi}$ durch jeweils genau eine Wertbelegung von z_1 und z_2 modelliert werden.

Φ und $\overline{\Psi}$ besitzen gemeinsame Variablen, und wir können nun den Verfeinerungsbeweis

$$\Phi \Rightarrow \overline{\Psi} \tag{3.48}$$

führen. Der Beweis wird in drei Schritte zerlegt, in denen die initiale Bedingung, die Zustandsfolgerelationen und die Fairnessbedingungen verifiziert werden. Der Beweis der initialen Bedingung

$$Init_\Phi \Rightarrow \overline{Init_\Psi} \tag{3.49}$$

ist trivial. Die Verifikation der Zustandsfolgerelationen ($[\pi_\Phi]_{w_\Phi} \Rightarrow [\overline{\pi_\Psi}]_{(z_1,z_2)}$) kann in Lemmata zerlegt werden, in denen jeweils nur eine einzelne Aktion bewiesen wird:

$$\begin{array}{lll} P_{1_\Phi} \Rightarrow \overline{P_{1_\Psi}} & P_{2_\Phi} \Rightarrow \overline{P_{2_\Psi}} & V_{1_\Phi} \Rightarrow \overline{V_\Psi} \\ V_{2_\Phi} \Rightarrow \overline{V_\Psi} & F_\Phi \Rightarrow Unchanged\ (z_1, z_2) & Unchanged\ w_\Phi \Rightarrow Unchanged\ (z_1, z_2) \end{array} \tag{3.50}$$

Der Beweis $V_{1_\Phi} \Rightarrow \overline{V_\Psi}$ kann nicht direkt geführt werden. In $\overline{V_\Psi}$ wird verlangt, daß die Variable z_2 nach Ausführung der Aktion den Wert 'aus' hat. Das kann jedoch nicht aus der Aktion V_{1_Φ} geschlossen werden, da sie nur zusichert, daß z_2 seinen Wert nicht verändert, jedoch keine Aussage über den tatsächlichen Wert macht. Zur Hilfe nehmen wir deshalb die bereits bewiesene Invariante I (siehe Formel 3.41) und können aufgrund der TLA-Regel INV2

$$V_{1_\Phi} \wedge I \wedge I' \Rightarrow \overline{V_\Psi} \tag{3.51}$$

beweisen. Da V_{1_Φ} nur schalten darf, wenn z_1 den Wert 'in' hat, muß z_2 nach I den Wert 'aus' besitzen. Daraus folgt jedoch $z_2' =$ 'aus' und der Verfeinerungsbeweis kann erbracht werden. Der Beweis der Aktion V_{2_Φ} erfolgt analog. Die anderen Beweise aus Formel 3.50 sind trivial. Aus den Lemmata in 3.50 folgt $[\pi_\Phi]_{w_\Phi} \Rightarrow [\overline{\pi_\Psi}]_{(z_1,z_2)}$. Mit den TLA-Regeln STL4 bzw. TLA2 (siehe Abbildung 3.2) kann man verifizieren, daß auch

$$\Box[\pi_\Phi]_{w_\Phi} \Rightarrow \Box[\overline{\pi_\Psi}]_{(z_1,z_2)} \tag{3.52}$$

gilt. Zuletzt müssen wir beweisen, daß Φ die Fairnessannahmen von $\overline{V_\Psi}$ erfüllt.

Fairnessbeweise erfordern zwar Deduktionen in temporaler Logik, können jedoch verhältnismäßig einfach auf Prädikate zurückgeführt werden. Beim Beweis der Zustandsfolgerelation haben wir gezeigt, daß zum Beispiel die Aktion P_{1_Φ} aus Φ die Aktion $\overline{P_{1_\Psi}}$ aus $\overline{V_\Psi}$ implementiert (siehe Formel 3.50). Wenn wir nun die starke Fairness von $\overline{P_{1_\Psi}}$ beweisen möchten, müssen wir zeigen, daß in allen von Φ modellierten Zustandsfolgen die Schaltbedingung von $\overline{P_{1_\Psi}}$ nur endlich oft schaltbar ist, oder daß Aktion P_{1_Φ} und damit $\overline{P_{1_\Psi}}$ unendlich oft schaltet. TLA unterstützt derartige Beweise durch die Ableitungsregeln SF2 und WF2 (siehe Abbildung 3.2). In diesem Beispiel ist der Beweis einfach, da P_{1_Φ} und $\overline{P_{1_\Psi}}$ identische Schaltbedingungen haben und P_{1_Φ} aufgrund seiner starken Fairness unendlich oft schalten muß, wenn die Aktion $\overline{P_{1_\Psi}}$ unendlich oft schaltbar ist. Formal erbringen wir diesen Beweis durch Verifikation der Prämissen von Regel SF2, in der π durch π_Φ, α und β durch P_{1_Φ}, $\overline{\lambda}$ durch $\overline{P_{1_\Psi}}$, F und T durch $True$, w durch w_Φ und $\overline{u}$ durch (z_1, z_2) ersetzt werden. Der Beweis der ersten Prämisse folgt aus dem Verfeinerungsbeweis in Formel 3.50. Die Verifikation der zweiten und dritten Prämisse ist trivial. Da $\Diamond\Box True$ sämtliche Zustandsfolgen aus ST^∞ modelliert, gilt die vierte Prämisse trivialerweise. Der Beweis der starken Fairness von Aktion $\overline{P_{2_\Psi}}$ wird analog geführt.

Etwas komplizierter ist der Beweis der schwachen Fairness von Aktion $\overline{V_\Psi}$, die auch schaltbar sein kann, wenn die sie realisierenden Aktionen V_{1_Φ} und V_{2_Φ} nicht geschaltet werden

dürfen. Wir zeigen dazu zunächst per TLA-Regel WF2, daß die schwache Fairness von Aktion $\overline{V_\Psi}$ durch die Fairness von V_{1_Φ} garantiert wird, wenn sichergestellt ist, daß die Schaltbedingung von V_{1_Φ} in jeder Zustandsfolge von Φ irgendwann erfüllt ist und nur durch Schalten von V_{1_Φ} wieder aufgehoben werden kann. Diese Annahme wird durch die vierte Prämisse der Regel WF2 beschrieben, in der π durch π_Φ, α und β durch V_{1_Φ}, $\overline{\lambda}$ durch $\overline{V_\Psi}$, F durch *Enabled* V_{1_Φ}, T durch Φ, w durch w_Φ und $\overline{u}$ durch (z_1, z_2) ersetzt sind. Der Beweis der drei ersten Prämissen ist trivial. Der Beweis der vierten Prämisse wird in zwei Lemmata zerlegt:

$$\Box[\pi_\Phi \wedge \neg V_{1_\Phi}]_{w_\Phi} \quad \Rightarrow \quad (\Diamond \mathit{Enabled}\ V_{1_\Phi} \Rightarrow \Diamond\Box \mathit{Enabled}\ V_{1_\Phi}) \tag{3.53}$$

$$\Phi \quad \Rightarrow \quad \Diamond \mathit{Enabled}\ V_{1_\Phi} \tag{3.54}$$

Der Teilbeweis von 3.53 wird erbracht, indem man zunächst *Enabled* $V_{1_\Phi} \Rightarrow \Box$*Enabled* V_{1_Φ} durch einen Stabilitätsbeweis (TLA-Regel INV1) verifiziert. Daraus folgt das Lemma 3.53, da aufgrund der TLA-Regel STL4 auch $(T \Rightarrow U) \Rightarrow (\Diamond T \Rightarrow \Diamond U)$ gilt.

Um den Beweis 3.54 führen zu können, definieren wir eine Folge an Zustandsprädikaten

$$(z_1 = \text{'in'} \wedge ka = \text{'aktiv'}) \quad \rightsquigarrow \quad \mathit{Enabled}\ V_{1_\Phi} \tag{3.55}$$

$$z_1 = \text{'aus'} \quad \rightsquigarrow \quad (z_1 = \text{'in'} \wedge ka = \text{'aktiv'}) \tag{3.56}$$

Wenn 3.55 und 3.56 wahr sind, kann man mit Hilfe der TLA-Regel LATTICE beweisen, daß jeder beliebige Zustand dieser Folge irgendwann in einen Zustand führt, in dem *Enabled* V_{1_Φ} gilt:

$$(z_1 \in \{\text{'aus'}, \text{'in'}\} \wedge ka \in \{\text{'aktiv'}, \text{'fertig'}\}) \rightsquigarrow \mathit{Enabled}\ V_{1_\Phi} \tag{3.57}$$

Da die Vorbedingung des $\rightsquigarrow$-Operators jedoch in Φ immer wahr ist, folgt daraus die Formel 3.54 und aufgrund der Stabilität (Formel 3.53) die vierte Prämisse der Regel WF2. Der Beweis von 3.55 wird mit Hilfe der TLA-Regel WF1 durchgeführt, in der π durch π_Φ, α durch F_Φ, F durch $(z_1 = \text{'in'} \wedge ka = \text{'aktiv'})$, G durch *Enabled* V_{1_Φ} und w durch w_Φ ersetzt werden. Die Prämissen können einfach bewiesen werden. Weniger trivial ist der Beweis von 3.56, wozu wir die TLA-Regel SF1 verwenden, in der π durch π_Φ, α durch P_{1_Φ}, F durch $z_1 = \text{'aus'}$, G durch $z_1 = \text{'in'} \wedge ka = \text{'aktiv'}$, T durch Φ und w durch w_Φ ersetzt werden. Der Beweis der ersten beiden Prämissen von Regel SF1 erfolgt wieder durch einfaches logisches Schließen. Der Beweis der dritten Prämisse

$$\Box(z_1 = \text{'aus'}) \wedge \Box[\pi_\Phi]_{w_\Phi} \wedge \Box\Phi \Rightarrow \Diamond \mathit{Enabled}\ P_{1_\Phi} \tag{3.58}$$

erfolgt hingegen analog zum Beweis von 3.54, wobei die Folgen an Zustandsprädikaten

$$(z_1 = \text{'aus'} \wedge z_2 = \text{'in'} \wedge ka = \text{'fertig'}) \quad \rightsquigarrow \quad \mathit{Enabled}\ P_{1_\Phi} \tag{3.59}$$

$$(z_1 = \text{'aus'} \wedge z_2 = \text{'in'}) \quad \rightsquigarrow \quad (z_1 = \text{'aus'} \wedge z_2 = \text{'in'} \wedge ka = \text{'fertig'}) \tag{3.60}$$

verwendet werden, die mit Hilfe der Regel WF1 bewiesen werden können. Daraus folgt jedoch nach der Regel LATTICE

$$(z_1 = \text{'aus'} \wedge z_2 \in \{\text{'aus'}, \text{'in'}\}) \rightsquigarrow \mathit{Enabled}\ P_{1_\Phi} \tag{3.61}$$

und daraus die Formel 3.58.

Nachdem wir nun aber die initiale Bedingung, die Zustandsfolgerelation und die Fairnessannahmen von Ψ aus Φ herleiten konnten, ist unsere Behauptung 3.48 bewiesen.

Das Vorgehen in diesem Beispiel ist typisch für Verfeinerungsbeweise in TLA (siehe auch die Beweise der Sätze 5.6 und 5.7 in Kapitel 5). Der Beweis der initialen Bedingung erfolgt durch einfaches logisches Schließen. Beim Beweis der Zustandsübergangsrelation wird unter Umständen mit Hilfe einer Invarianten gezeigt, daß alle Aktionen des Feinsystems Φ jeweils eine Aktion oder einen Stotterschritt des Grobsystems Ψ abbilden. Diese Abhängigkeit zwischen den Aktionen von Fein- und Grobsystem wird auch für die Fairnessbeweise verwendet. Mit Hilfe der TLA-Regeln WF2 oder SF2 beweist man, daß aus der Fairness einer Aktion des Feinsystems die Fairness der entsprechenden Aktion des Grobsystems folgt, wenn entweder die Aktion des Grobsystems häufig genug nicht schaltbar ist oder die Aktion des Feinsystems ab einem bestimmten Zeitpunkt immer schaltbar ist, was durch die Temporalformel $\Diamond\Box F$ beschrieben wird (F ist ein Prädikat des Feinsystems). Der Beweis von $\Diamond\Box F$ erfolgt durch einen Stabilitätsbeweis $\Diamond F \leadsto \Diamond\Box F$ und einen temporalen Beweis von $\Diamond F$. Für den temporalen Beweis wird eine Folge von Zustandsprädikaten

$$F_0 \leadsto F_1 \leadsto \ldots \leadsto F_n \equiv F$$

aufgestellt, für die

$$\Phi \Rightarrow \Box(\exists i : F_i)$$

gilt. Die einzelnen Zustandsübergänge $F_i \leadsto F_{i+1}$ werden mit Hilfe der TLA-Regeln SF1 oder WF1 bewiesen. Beim Beweis mit Hilfe der Regel SF1 muß eine temporale Prämisse der Art $\Diamond G$ bewiesen werden, während in WF1 nur einfache prädikatenlogische Verifikationen vorkommen. Der Beweis von $\Diamond G$ erfolgt „rekursiv“ durch Aufstellen und Beweisen neuer Folgen von Zustandsprädikaten. Die „Rekursion“ terminiert, wenn alle Zustandsübergänge mit Hilfe der TLA-Regel WF1 bewiesen werden können.

3.3.4 Refinement Mappings

In [AL91a] schlagen Abadi und Lamport eine zu logischen Deduktionen alternative Methode für Verfeinerungsbeweise vor. Wie wir in Abschnitt 3.3.2 zeigten, beschreibt man Safetyeigenschaften von Systemen durch Zustandstransitionssysteme. Diese Automaten werden zu Zustandstransitionssystemen der Art (Σ, F, Δ, L) erweitert. $\Sigma \subseteq \Sigma_E \times \Sigma_I$ ist die Menge an Zuständen, die sich aus nach außen sichtbaren Zustandskomponenten (aus Σ_E) und nach außen unsichtbaren Zustandskomponenten (aus Σ_I) zusammensetzen. $F \subseteq \Sigma$ beschreibt eine Menge von Anfangszuständen und $\Delta \subseteq (\Sigma \times \Sigma)$ eine reflexive Zustandsübergangsrelation. L ist eine Menge von stotterinvarianten Zustandsfolgen. Die Safetyeigenschaften einer Spezifikation Φ werden durch die durch (Σ, F, Δ) definierten Zustandsfolgen M („Machine Property“) modelliert. Die Livenesseigenschaften werden durch die Zustandsfolgen aus L („Supplementary Property“) beschrieben. Das Verhalten P_Φ, die Menge aller in Φ erlaubten Zustandsfolgen, wird durch

$$P_\Phi \triangleq M \cap L \tag{3.62}$$

gebildet.

Abadi und Lamport haben bewiesen, daß eine Spezifikation $\Phi = (\Sigma_\Phi, F_\Phi, \Delta_\Phi, L_\Phi)$ eine Verfeinerung einer Spezifikation $\Psi = (\Sigma_\Psi, F_\Psi, \Delta_\Psi, L_\Psi)$ ist, wenn ein Refinement Mapping

(eine Zustandsabbildung f mit einigen zusätzlichen Eigenschaften) existiert. Dazu wird angenommen, daß Φ und Ψ identische nach außen sichtbare Zustandskomponenten ($\Sigma_{\Phi_E} = \Sigma_{\Psi_E}$) besitzen. Weiterhin führen wir die Abbildung $S_E : \Sigma_E \times \Sigma_I \to \Sigma_E$ ein, die einen Zustand auf seine externe Zustandskomponente abbildet.

Unter einem Refinement Mapping f verstehen wir eine Abbildung

$$f : \Sigma_\Phi \to \Sigma_\Psi \tag{3.63}$$

vom Zustandsraum von Φ auf den Zustandsraum von Ψ, die die folgenden Nebenbedingungen erfüllt:

R1: $\forall s \in \Sigma_\Phi : S_E(f(s)) = S_E(s)$ (f bildet nach außen sichtbare Zustandskomponenten von Φ auf die identischen Zustandskomponenten in Ψ ab)

R2: $\forall s : s \in F_\Phi \Rightarrow f(s) \in F_\Psi$ (f überführt die initialen Zustände von Φ auf initiale Zustände in Ψ)

R3: $\forall s, t : \langle s, t\rangle \in \Delta_\Phi \Rightarrow ((\langle f(s), f(t)\rangle \in \Delta_\Psi) \vee (f(s) = f(t)))$ (f bildet Zustandsübergänge von Φ auf Zustandsübergänge oder auf Stotterschritte in Ψ ab)

R4: $f(P_\Phi) \subseteq L_\Psi$ (f bildet die Zustandsfolgen aus P_Φ auf Zustandsfolgen der „Supplementary Property“ von Ψ ab[6])

Beim Beweis der Bedingungen R1, R2 und R3 müssen jeweils nur einzelne Zustände bzw. Zustandspaare untersucht werden, während für R4 ganze Zustandsfolgen untersucht werden.

Die Existenz eines Refinement Mappings ist eine hinreichende, jedoch nicht notwendige Bedingung für Verfeinerungen. Zum Beispiel gibt es keine Refinement Mappings zwischen zwei Systemen Φ und Ψ, wenn Ψ im Gegensatz zu Φ die Anzahl der bisher ausgeführten Zustandsübergänge gespeichert hat. Dann existiert keine Abbildung f von Σ_Φ auf Σ_Ψ. Eine derartige Situation existiert auch beim Framework für Hochleistungsprotokolle (siehe Abschnitt 5.7). Man findet ebenfalls keine Refinement Mappings, wenn in Ψ indeterministische Entscheidungen früher als in Φ getroffen werden oder wenn Φ in Teilen abstrakter als Ψ modelliert wird. Allerdings kann man in diesen Fällen ein Refinement Mapping für eine erweiterte Spezifikation Φ^{hp} finden. Φ^{hp} wird aus Φ erzeugt, indem man zusätzliche Vergangenheitsvariablen („History Variables“) h und Zukunftsvariablen („Prophecy Variables“) p einführt. Vergangenheitsvariablen speichern Informationen über die Historie von Systemen. Zukunftsvariablen erlauben, indeterministische Entscheidungen vorzuziehen. Φ^{hp} besitzt das gleiche nach außen sichtbare Verhalten wie Φ (vgl. [AL91a]):

$$\exists i : \Phi \Rightarrow \exists i, h, p : \Phi^{hp} \tag{3.64}$$

Abadi und Lamport weisen nach, daß, wenn Φ eine Verfeinerung von Ψ ist, aus Φ eine erweiterte Spezifikation Φ^{hp} gebildet werden kann, für die ein Refinement Mapping f zwischen $\Sigma_{\Phi^{hp}}$ und Σ_Ψ existiert. Dazu müssen Φ^{hp} und Ψ allerdings einige weitere Eigenschaften erfüllen. In Φ^{hp} darf die Zustandsmenge $L_{\Phi^{hp}}$ keine zusätzlichen Safetyeigenschaften modellieren („Machine Closure“). Das ist garantiert, wenn man zur Beschreibung der Liveness nur

[6] Unter $f(P_\Phi)$ verstehen wir die Abbildung, in der jeder Zustand s in jeder Zustandsfolge aus P_Φ durch den Zustand $f(s)$ ersetzt wird.

starke oder schwache Fairness verwendet. Außerdem muß die Menge aller Zustandsübergänge, deren externe Zustandskomponente denselben Übergang aufweist, endlich sein („Finite Invisible Nondeterminism"). Weiterhin muß garantiert sein, daß Zustandsfolgen, die nicht in Ψ erlaubt sind, nur durch Betrachtung der externen Komponenten und einer endlichen Zahl an Übergängen des internen Verhaltens entdeckt werden können („Internal Continuity"). Unter Einhaltung dieser Eigenschaften kann man immer ein Refinement Mapping für den Beweis

$$\Phi^{hp} \Rightarrow \Psi \tag{3.65}$$

finden.

Das Refinement Mapping-Konzept ermöglicht die Verwendung von Werkzeugen für Verfeinerungsbeweise, die auf Model Checking basieren (siehe Abschnitt 3.5).

3.3.5 Komposition

Eine Spezifikation Φ modelliert nur das Verhalten der Variablen v_Φ, deren Bezeichner in Φ enthalten sind. Über alle anderen Variablen aus Var werden in Φ keine Aussagen gemacht. Daraus folgt, daß man zwei Subsysteme Φ_1 und Φ_2, die keine gemeinsamen Variablen besitzen, durch Konjunktion zu einem System Φ nebenläufig koppeln kann (siehe [Lam94]):

$$\Phi \triangleq \Phi_1 \wedge \Phi_2 \tag{3.66}$$

In [AL93a, AL93b] stellen Abadi und Lamport ein Kompositionsprinzip für Subsysteme zu gekoppelten Systemen vor, die miteinander durch gemeinsam genutzte Variablen interagieren (siehe Abschnitt 3.1.4). Das interne Verhalten eines Systems und das von seiner Umgebung geforderte Verhalten werden durch zwei TLA-Formeln Φ^S bzw. Φ^U modelliert, die im kanonischen TLA-Stil formuliert sind. Das aus internem Systemverhalten und dem Verhalten der Systemumgebung bestehende Verhalten des Gesamtsystems, das durch Φ modelliert wird, wird durch eine Implikation beschrieben:

$$\Phi \triangleq \Phi^U \Rightarrow \Phi^S \tag{3.67}$$

Abadi und Lamport vertreten dabei den Standpunkt, daß ein System niemals die Eingaben seiner Umgebung beeinflussen kann. Man sichert korrektes Verhalten Φ^S nur dann zu, wenn sich die Systemumgebung an die Verhaltensanforderung Φ^U hält.

Ziel einer Kompositionsverifikation ist der Nachweis, daß die Konjunktion der Subsysteme $(\Phi_i^U \Rightarrow \Phi_i^S)$ das durch $\Phi^U \Rightarrow \Phi^S$ modellierte Gesamtsystem erfüllt:

$$(\bigwedge_{i=1}^{n} (\Phi_i^U \Rightarrow \Phi_i^S)) \Rightarrow (\Phi^U \Rightarrow \Phi^S) \tag{3.68}$$

An die Komposition werden dabei die folgenden Bedingungen gestellt, deren Gültigkeit bewiesen werden muß (vgl. [AL93a]):

- Das Gesamtsystem erfüllt Φ^S dann, wenn die Teilsysteme Φ_i^S erfüllen.
- Die Teilsysteme erfüllen Φ_i^S, wenn ihre Umgebungen Φ_i^U erfüllen. Diese Umgebungen der Teilsysteme bestehen aus der Umgebung des Gesamtsystems, die durch Φ^U modelliert wird, und aus allen anderen Teilsystemen.

Man sieht sofort, daß die Schlußfolge zirkular ist. Die Umgebungen der Teilsysteme erfüllen Φ_i^U nur, wenn die Teilsysteme Φ_i^S erfüllen. Die Teilsysteme Φ_i^S werden jedoch nur erfüllt, wenn ihre Umgebungen Φ_i^U erfüllen.

Abadi und Lamport haben jedoch bewiesen, daß man das Kompositionsprinzip dennoch anwenden kann, wenn die Spezifikation der Systemumgebung Φ_i^U keine Livenessanforderungen enthält. Bevor wir das stärkere Kompositionsprinzip beschreiben, erläutern wir drei neue temporale Operatoren, die aus $'$, $\Box$ und $\exists$ abgeleitet werden können (vgl. [AL93b]):

- Die Hülle $\mathcal{C}(\Phi)$ einer Spezifikation Φ ist die Beschreibung der Safetyeigenschaften von Φ. Wenn
$$\Phi \triangleq Init_\Phi \wedge \Box[\pi_\Phi]_{v_\Phi} \wedge L_\Phi$$
eine kanonische Beschreibungsform hat, dann wird ihre Hülle durch
$$\mathcal{C}(\Phi) \triangleq Init_\Phi \wedge \Box[\pi_\Phi]_{v_\Phi}$$
beschrieben.

- Der Operator T_{+v} sagt aus, daß die Variable v nur neue Wertbelegungen annehmen darf, wenn die temporale Formel T in allen vorhergehenden Zuständen erfüllt war:
$$\langle s_0, s_1, s_2, \ldots\rangle[\![T_{+v}]\!] \triangleq \forall n, m \in Nat : (m \leq n \Rightarrow s_m[\![T]\!]) \vee s_n[\![v]\!] = s_{n+1}[\![v]\!]$$

- Durch den Operator $T \overset{+}{\hookrightarrow} U$ wird beschrieben, daß U für mindestens einen Schritt länger erfüllt ist als T:
$$\langle s_0, s_1, s_2, \ldots\rangle[\![T \overset{+}{\hookrightarrow} U]\!] \triangleq \forall n, m \in Nat : m < n \wedge s_m[\![T]\!] \Rightarrow s_n[U] \wedge s_m[U]$$

Die Beschreibung $\Phi^U \overset{+}{\hookrightarrow} \Phi^S$ ist im Prizip stärker als $\Phi^U \Rightarrow \Phi^S$, weil sie Zustandsfolgen ausschließt, in denen Φ^U und Φ^S im gleichen Zustand zum erstem Mal ungültig werden. In realisierbaren Systemen können Φ^U und Φ^S jedoch niemals gleichzeitig zum ersten Mal ungültig werden, da ein System nicht „wissen“ kann, ob seine Umgebung im nächsten Schritt ihre Anforderung erfüllt oder nicht, weil es selbst keinen Einfluß auf das Verhalten seiner Umgebung hat. Folglich muß es sich konform zu seiner Anforderung Φ^S verhalten, wenn die Umgebung bisher immer Φ^U erfüllt hat. In der Praxis beschreiben also $\Phi^U \Rightarrow \Phi^S$ und $\Phi^U \overset{+}{\hookrightarrow} \Phi^S$ das gleiche Verhalten.

Das Kompositionstheorem für die Kopplung eines Systems $\Phi^U \Rightarrow \Phi^S$ aus n Subsystemen $\Phi_i^U \Rightarrow \Phi_i^S$ wird von Abadi und Lamport folgendermaßen definiert (vgl. [AL93b]):

1. $\models \forall i \in \{1, \ldots, n\} : \mathcal{C}(\Phi^U) \wedge \bigwedge_{j=1}^{n} \mathcal{C}(\Phi_j^S) \Rightarrow \Phi_i^U$

2. (a) $\models \mathcal{C}(\Phi^U)_{+v} \wedge \bigwedge_{j=1}^{n} \mathcal{C}(\Phi_j^S) \Rightarrow \mathcal{C}(\Phi^S)$

   (b) $\models \Phi^U \wedge \bigwedge_{j=1}^{n} \Phi_j^S \Rightarrow \Phi^S$

then $\models \bigwedge_{j=1}^{n} (\Phi_j^U \overset{+}{\hookrightarrow} \Phi_j^S) \Rightarrow (\Phi^U \overset{+}{\hookrightarrow} \Phi^S)$

Es stellt sichert, daß ein System durch die Komposition von Subsystemen realisiert wird, wenn aus der Umgebungsbedingung Φ^U des Gesamtsystems sowie der Konjunktion aller Subsystembeschreibungen Φ_j^S sowohl die Umgebungsbedingungen Φ_j^U der Subsysteme als auch die Verhaltensbeschreibung Φ^S des Gesamtsystems hergeleitet werden können. Durch das Theorem wird impliziert, daß die Umgebungsbedingungen der Subsysteme Φ_i^U keine Livenessanforderungen enthalten dürfen, da die Formel 1 sonst nicht bewiesen werden könnte.

Wie bereits in Abschnitt 3.1.4 dargelegt wurde, baut die hier vorgestellte Kompositionsmethode auf gemeinsam genutzten Variablen auf und ist deshalb nicht sehr gut für constraintorientierte Spezifikationen geeignet.

3.4 TLA$^+$ und tTLA$^+$

Wie in Abschnitt 3.3.2 gezeigt wurde, spezifiziert man Systeme in TLA durch temporallogische Formeln in kanonischer Form. Da diese Formeln bei realistischen Systemen sehr lang werden können, ist eine über die in den Abbildungen 3.1 und 3.3 beschriebene TLA-Syntax hinausgehende Spezifikationssprache notwendig. TLA-Formeln müssen in Subformeln zerlegbar sein, die man auf unterschiedliche Module aufteilen kann. Mit TLA$^+$ [Lam95] bietet Lamport eine derartige Syntax an. Allerdings kann man mit den üblichen Übersetzerbauwerkzeugen keine Übersetzer entwickeln, die TLA$^+$ als Eingabesprache besitzen (vgl. [Hey95]). Die Arbeitsgruppe „Rechnernetze und verteilte Systeme“ an der Universität Dortmund hat sich deshalb entschlossen, eine eigene Variante tTLA$^+$ (tool TLA$^+$) [Mes93] zu entwerfen, die Generatoren von Shift/Reduce-Parsern (vgl. [WG84]) unterstützt. Im folgenden geben wir einen kurzen Überblick über TLA$^+$ und skizzieren anschließend kurz die Änderungen in tTLA$^+$. Für eine ausführlichere Betrachtung seien [Lam95, Mes93, Hey95] empfohlen.

3.4.1 TLA$^+$

Module bilden in TLA$^+$ die kleinste strukturelle Einheit. Sie bestehen aus den folgenden Teilen:

- Modulname
- Definitionen von Operatoren
- Parameterdeklarationen
- Logische Annahmen
- Theoreme
- Methoden zur Einbeziehung anderer Module und zum Export von Definitionen

In Abbildung 3.4 stellen wir eine leicht modifizierte Beschreibung der Spezifikation Φ (Formel 3.39) als Beispiel für eine TLA$^+$-Spezifikation vor, die den Namen `FSys` trägt. Die waagerechten Striche dienen der Übersichtlichkeit und haben keine weitere Bedeutung.

Die in Spezifikationen verwendeten Parameter und Definitionen werden in Klassen, sogenannte Sorten, eingeteilt. Die Sorten hängen dabei davon ab, ob Konstanten, Variablen, Variablen mit $'$-Symbol („Prime“), oder temporale Operatoren verwendet werden dürfen (siehe Abschnitt 3.3.1):

MODULE FSys

IMPORT Nat *Import des Datentyps Nat*

CONSTANT
Ptyp $\triangleq$ {1, 2} *Indizes der um den kritischen Abschnitt konkurrierenden Prozesse*

PARAMETERS z : [Ptyp $\rightarrow$ {' $\vee -\vee$ 'aus″, ' $\vee -\vee$ 'in″}] VARIABLE *Prozess Pi im kritischen Abschnitt?*
ka : {' $\vee -\vee$ 'fertig″, ' $\vee -\vee$ 'aktiv″} VARIABLE *Prozess im kritischen Abschnitt noch aktiv?*

STATE PREDICATE
Init $\triangleq$ z = [i $\in$ Ptyp $\mapsto$ ' $\vee -\vee$ 'aus″] $\wedge$ ka = ' $\vee -\vee$ 'fertig″ *Initialbedingung*

ACTION
P (p : Ptyp) $\triangleq$ z = [i $\in$ Ptyp $\mapsto$ ' $\vee -\vee$ 'aus″] $\wedge$ z′ = [z Except ![p] = ' $\vee -\vee$ 'in″] $\wedge$ ka′ = ' $\vee -\vee$ 'aktiv″
P-Prozedur des Semaphors
V (p : Ptyp) $\triangleq$ z[p] = ' $\vee -\vee$ 'in″ $\wedge$ ka = ' $\vee -\vee$ 'fertig″ $\wedge$ z′ = [z Except ![p] = ' $\vee -\vee$ 'aus″] $\wedge$ Unchanged ka
V-Prozedur des Semaphors
F $\triangleq$ ka = ' $\vee -\vee$ 'aktiv″ $\wedge$ ka′ = ' $\vee -\vee$ 'fertig″ $\wedge$ Unchanged (z_1,z_2) *Beendigung der Arbeit im krit. Ab.*
$\pi \triangleq \exists$ p $\in$ Ptyp : P(p) $\vee \exists$ p $\in$ Ptyp : V(p) $\vee$ F *Zustandsfolgerelation*

STATE FUNCTION
var $\triangleq$ (z,ka) *Tupel der Variablen im System*

STATE PREDICATE
Typ $\triangleq$ z $\in$ [Ptyp $\rightarrow$ {' $\vee -\vee$ 'aus″, ' $\vee -\vee$ 'in″}] $\wedge$ ka $\in$ {' $\vee -\vee$ 'fertig″, ' $\vee -\vee$ 'aktiv″} *Datentypinvariante*
Inv $\triangleq \forall$ p $\in$ Ptyp : (z[p] $\neq$ ' $\vee -\vee$ 'aus″ $\Rightarrow (\forall$ q $\in$ Ptyp $\setminus$ {p} : z[q] = ' $\vee -\vee$ 'aus″))
Invariante: Nur maximal ein Prozess im kritischen Abschnitt

TEMPORAL FORMULA
Live $\triangleq \forall$ p $\in$ Ptyp : $SF_{\texttt{var}}$(P(p)) $\wedge \forall$ p $\in$ Ptyp : $WF_{\texttt{var}}$(V(p)) $\wedge$ $WF_{\texttt{var}}$(F) *Fairnessanforderung*
Sys $\triangleq$ Init $\wedge$ $\Box[\pi] \wedge$ Live *Systemformel*

THEOREM
Sys $\Rightarrow \Box$ (Typ $\wedge$ Inv) *Theorem: System erfuellt die Invarianten*

Abbildung 3.4: TLA$^+$-Spezifikation des Semaphors Φ

Constant (Term) / Boolean (Formel): Ausdruck, der nur Bezeichner von Konstanten enthält (z.B. $\{1, 2\}$; 'Nosi' $\neq$ 6)

State Function (Term) / State Predicate (Formel): Ausdruck, der Variablen ohne $'$-Symbol und Konstanten enthält (z.B. (z, ka); $ka =$ 'fertig')

Transition Function (Term) / Action (Formel): Ausdruck, der Konstanten, Variablen

und Variablen mit $'$-Symbol enthält (z.B. (z, ka'); $ka' = $ 'fertig')

Temporal Formula Temporalformel, die Konstanten, Variablen, Variablen mit $'$-Symbol und temporale Operatoren enthält (z.B. $\Box[\pi] \wedge WF_{var}(F)$)

Parameter eines Moduls können Variablen, konstante Terme oder Boolesche Formeln sein. Da alle Variablenbezeichner, die in TLA-Formeln auftreten können, aus der globalen Menge Var stammen, ist nie sichergestellt, daß eine Variable nur in einer einzigen TLA-Formel referenziert und in ihrer Verhaltensweise beschränkt wird. Lamport verzichtet deshalb in TLA$^+$ auf einen Variablendeklarationsteil, der implizieren würde, daß Variablen nur in diesem lokalen Modul referenziert werden können. Stattdessen führt er Variablen als Modulparameter ein (z.B. z und ka in Abbildung 3.4) und stellt damit sicher, daß Variablen immer von der Modulumgebung gelesen werden können. Der Sortenbezeichner `VARIABLE` bezeichnet Variablenparameter. Konstante oder Boolesche Ausdrücke werden durch die Sortenbezeichner `CONSTANT` bzw. `BOOLEAN` gekennzeichnet. Da Variablen beliebige Werte aus der Menge Val annehmen dürfen, sind Typvereinbarungen von Variablen nicht sinnvoll. TLA$^+$ erlaubt jedoch, Parameter von Variablen bzw. Termen mit der Angabe einer Menge erlaubter Werte zu versehen, die nur als ein Kommentar verstanden werden soll (z.B. `{ka : "fertig", "aktiv"}`). Der Nachweis, daß die Bezeichner tatsächlich nur Werte aus dieser Menge annehmen, muß mit einem eigenen Invariantenbeweis geführt werden.

Bei der Definition von Operatoren wird zunächst die Sorte des Operators durch ein Schlüsselwort wie `ACTION` festgelegt. Am Anfang einer Definition steht der Name des Operators (z.B. `P`). Ein Operator bildet ein Tupel von Ausdrücken auf einen einfachen Ausdruck ab. Der Definitionsbereich dieser Abbildung wird durch eine Parameterliste aus einfachen Bezeichnern (`p`) angegeben, in der auch ein Typ angegeben werden darf (z.B. `p : Ptyp`). Bei konstanten Abbildungen kann die Parameterliste weggelassen werden (z.B. Aktion `F`). Eine Operatordefinition eines Terms kann ebenfalls mit einer Typangabe versehen werden. Hinter dem Schlüsselwort $\triangleq$ wird die Formel angegeben, die den Operator definiert (z.B. `z` $\triangleq$ `[i ∈ Ptyp ↦ "aus"]`).

Die Syntax der Terme und Ausdrücke orientiert sich an bekannten Programmier- und Spezifikationssprachen wie Modula [Wir85] und Z [Spi89]. Die temporalen Operatoren wurden im wesentlichen schon in Abschnitt 3.3 vorgestellt. Ein weiterer Operator $\pi \cdot \lambda$ beschreibt die Aneinanderreihung zweier Aktionen π und λ:

$$s[\![\pi \cdot \lambda]\!]t \triangleq \exists u \in \mathsf{St} : s[\![\pi]\!]u \wedge u[\![\lambda]\!]t \tag{3.69}$$

Der Operator $\forall$ wird durch

$$\forall x : T \triangleq \neg\exists x : \neg T \tag{3.70}$$

definiert und beschreibt universelle Quantifikation. Er bildet das Komplement zu dem in Abschnitt 3.3.1 definierten Operator $\exists$ („Bold Exists“). Eine Zusammenfassung der temporalen Operatoren von TLA$^+$ findet sich in Abbildung 3.5. Konstante Operatoren erlauben die Beschreibung logischer Abhängigkeiten, Mengen, Funktionen, Felder, Tupel, Zeichenketten und Nummern. Eine Auflistung dieser Operatoren findet sich in Abbildung 3.6.

Wir betrachten nun in Abbildung 3.7 ein zweites Modul `Sys`, das zur Beschreibung des Refinement Mappings `RM` und des Verfeinerungsbeweises $\Phi \Rightarrow \Psi$ dient. Die in Modul `FSYS` beschriebene Spezifikation Φ und die hier nicht aufgeführte TLA$^+$-Spezifikation von Ψ werden dazu inkludiert.

Predicate and Action Operators

x'	[x true in final state of step]
$[\pi]_w$	$[\pi \vee (w' = w)]$
$\langle\pi\rangle_w$	$[\pi \wedge (w' \neq w)]$
$Enabled\ \pi$	[An π step is possible]
$Unchanged\ w$	$[w' = w]$
$\pi \cdot \lambda$	[Composition of actions]

Temporal Operators

$\Box T$	[T is always true]
$\Diamond T$	[Eventually: $\neg\Box\neg T$]
$\mathrm{WF}_w(\pi)$	[Weak fairness: $\Box\Diamond\langle\pi\rangle_w \vee \Box\Diamond\neg Enabled\ \pi$]
$\mathrm{SF}_w(\pi)$	[Strong fairness: $\Box\Diamond\langle\pi\rangle_w \vee \Diamond\Box\neg Enabled\ \pi$]
$T \rightsquigarrow U$	[Leads to: $\Box(T \Rightarrow \Diamond U)$]
$\exists x : T$	[Temporal existential quantification (hiding).]
$\forall x : T$	[Temporal universal quantification.]

Abbildung 3.5: Temporale TLA$^+$-Operatoren (entnommen aus [Lam95] mit geänderten Bezeichnern)

Lamport kombiniert mit seinem Modulkonzept Spezifikationen und Verifikationen. Außer Parametern und Operatordefinitionen können Module auch logische Annahmen und logische Theoreme enthalten. Lamport bezeichnet ein Modul als gültig, wenn man die Theoreme (z.B. Φ`.Sys` $\Rightarrow \Psi$`.Sys`) aus den Annahmen (z.B. Φ`.Sys` $\Rightarrow \Box$ `(`Φ `.Typ` $\wedge \Phi$`.Inv)`) folgern kann.

Durch Exportbeschreibungen (z.B. `EXPORT RM`) können die Operatordefinitionen, die anderen Modulen zur Verfügung gestellt werden, eingeschränkt werden. Weiterhin kann durch den Befehl `EXPORT OPAQUE RM` nur der Export von Operatornamen ohne zugehörige Definitionen zugelassen werden. In unseren Beispielen sind Exportbeschränkungen nicht sinnvoll, da für die Theorembeweise alle Operatoren notwendig sind. Modulimport und Modulinklusion sind die Methoden zur Integration anderer Module. Beim Modulimport (z.B. `IMPORT Nat`) werden dem importierenden Modul alle für den Export zugelassenen Operatordefinitionen, sowie alle Parameter, Annahmen und Theoreme hinzugefügt. Die Symbole behalten ihre Namen. Bei der Inklusion hingegen werden die formalen Parameter des inkludierten Moduls durch Ausdrücke ersetzt. In `INCLUDE GSys AS` Ψ `WITH x` $\leftarrow$ `RM(z)` wird die Variable `x` aus Modul `GSys` durch den Ausdruck `RM(z)` ersetzt. Ψ gibt den Namen der Modulinstanz an. Mit Ausnahme von Variablen, die durch Zustandsfunktionen ersetzt werden dürfen, müssen die Sorten der Parameter eingehalten werden. Weiterhin werden alle Operatornamen des inkludierten Moduls umbenannt. Man fügt den Modulinstanznamen als Präfix hinzu (z.B. wird `Sys` in `FSYS` durch Φ`.Sys` ersetzt). Annahmen und Theoreme werden nicht inkludiert.

3.4.2 tTLA$^+$

TLA$^+$ sieht nicht nur die Definition von Operatoren in Präfix-Schreibweise vor, sondern auch von Infix-, Postfix- oder gemischten (Mixfix-) Operatoren. tTLA$^+$ [Mes93] erlaubt hingegen nur die Verwendung von Präfix-Operatoren und binären Infix-Operatoren. Um den Benutzer bei der Beschreibung komplexer arithmetischer Ausdrücke zu unterstützen, führt tTLA$^+$ ein

Logic

$True$ $False$ $\wedge$ $\vee$ $\neg$ $=>$ $\equiv$
$\forall\, x : p$ $\exists\, x : p$
$\forall\, x \in S : p$ $\exists\, x \in S : p$
$Choose\ x : p$ [Equals some x satisfying p]

Sets

$=$ $\neq$ $\in$ $\notin$ $\cup$ $\cap$ $\subseteq$: [set difference]
$\{e_1, \ldots, e_n\}$ [Set consisting of elements e_i]
$\{x \in S : p\}$ [Set of elements x in S satisfying p]
$\{e : x \in S\}$ [Set of elements e such that x in S]
$Subset\ S$ [Set of subsets of S]
$Union\ S$ [Union of all elements of S]

Functions

$f[e]$	[Function application]
$Domain\ f$	[Domain of function f]
$[x \in S \mapsto e]$	[Function f such that $f[x] = e$ for $x \in S$]
$[S \to T]$	[Set of functions f with $f[x] \in T$ for $x \in S$]
$[f\ Except\ ![e_1] = e_2]$	[Function $\widehat{f}$ equal to f except $\widehat{f}[e_1] = e_2$]
$\{[f\ Except\ ![e] \in S]\}$	[Set of functions $\widehat{f}$ equal to f except $\widehat{f}[e] \in S$]

Records

$e.h$	[The h-component of record e]
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	[The record whose h_i component is e_i]
$[h_1 : S_1, \ldots, h_n : S_n]$	[Set of all records with h_i component in S_i]
$[r\ Except\ !.h = e]$	[Record $\widehat{r}$ equal to r except $\widehat{r}.h = e$]
$\{[r\ Except\ !.h \in S]\}$	[Set of records $\widehat{r}$ equal to r except $\widehat{r}.h \in S$]

Tuples

$e[i]$	[The i^{th} component of tuple e]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with i^{th} component in S_i]

Strings and Numbers

'$c_1 \ldots c_n$'	[A literal string of n characters]
$String$	[The set of all strings]
$d_1 \ldots d_n$ $d_1 \ldots d_n.d_{n+1} \ldots d_m$	[Numbers]

Miscellaneous

if p **then** e_1 **else** e_2	[Equals e_1 if p true, else e_2]
case $p_1 \to e_1, \ldots, p_n \to e_n$	[Equals e_i if p_i true]
let $x_1 \triangleq e_1 \ \ldots \ x_n \triangleq e_n$ **in** e	[Equals e in the context of the definitions]

$\wedge\ p_1$	$\vee\ p_1$
$\ldots$ [the conjunction $p_1 \wedge \ldots \wedge p_n$]	$\ldots$ [the disjunction $p_1 \vee \ldots \vee p_n$]
$\wedge\ p_n$	$\vee\ p_n$

Abbildung 3.6: Konstante TLA$^+$-Operatoren (entnommen aus [Lam95] mit geänderten Bezeichnern)

vordefiniertes Modul für die natürlichen Zahlen einschließlich deren Operatoren und Axiomen ein. Weiterhin müssen wichtige Klauseln wie Operator-Definitionen, Parameter oder Theoreme voneinander durch Semikolons abgetrennt werden. Einige weitere kleinere syntaktische Änderungen, die in [Hey95] nachzulesen sind, vereinfachen die automatische Entwicklung von Übersetzern.

MODULE Sys

IMPORT Nat *Import des Datentyps Nat*

CONSTANT
Ptyp $\triangleq$ { 1, 2 } *Indizes der um den kritischen Abschnitt konkurrierenden Prozesse*

PARAMETERS z : [Ptyp $\rightarrow$ {' $\vee -\vee$ 'aus″, ' $\vee -\vee$ 'in″}] VARIABLE *Prozess Pi im kritischen Abschnitt?*
ka : { ' $\vee -\vee$ 'fertig″, ' $\vee -\vee$ 'aktiv″ } VARIABLE *Prozess im kritischen Abschnitt noch aktiv?*

STATE FUNCTION
RM (a) $\triangleq$ CASE a = [i $\in$ Ptyp $\mapsto$ ' $\vee -\vee$ 'aus″] $\rightarrow$ ' $\vee -\vee$ 'leer″, *Refinement Mapping*
a[1] = ' $\vee -\vee$ 'in″ $\wedge$ a[2] = ' $\vee -\vee$ 'aus″ $\rightarrow$ 1,
a[1] = ' $\vee -\vee$ 'aus″ $\wedge$ a[2] = ' $\vee -\vee$ 'in″ $\rightarrow$ 2,
OTHERWISE $\rightarrow$ ' $\vee -\vee$ 'illegal″

INCLUDE FSys AS Φ *Inklusion der Feinspezifikation*
INCLUDE GSys AS Ψ WITH x $\leftarrow$ RM(z) *Inklusion der Grobspezifikation*

ASSUMPTION
Φ.Sys $\Rightarrow$ $\Box$ (Φ.Typ $\wedge$ Φ.Inv) *Annahme: Feinspezifikation erfuellt ihre Invarianten*

THEOREM
Φ.Sys $\Rightarrow$ Ψ.Sys *Theorem: Feinspezifikation implementiert Grobspezifikation*

Abbildung 3.7: TLA$^+$-Spezifikation des Verfeinerungsbeweises zwischen den Semaphoren Φ und Ψ

3.5 Werkzeugunterstützung

Aufgrund der Komplexität verteilter Systeme ist eine Unterstützung ihres Entwurfs durch rechnergestützte Werkzeuge sinnvoll. Im Rahmen des Projekts „Tools for Transition Axioms Based Specifications“[7] wurden durch die Arbeitsgruppe „Rechnernetze und verteilte Systeme“ an der Universität Dortmund Werkzeuge für die Entwicklung formaler TLA-Spezifikationen und deren Verifikation entworfen und prototypisch implementiert (vgl. [MH94]).

3.5.1 Unterstützung der Spezifikationsentwicklung

Interessant bei der Entwicklung von Spezifikationen sind zum einen Werkzeuge zur Manipulation und zur Verwaltung von Spezifikationseinheiten wie TLA$^+$-Modulen. Zum anderen kann die Entwicklung komplexerer Systeme aber auch durch Programme zur symbolischen oder realen Ausführung von Spezifikationen wirkungsvoll unterstützt werden.

Zur Entwicklung von Spezifikationen steht ein syntax-orientierter Editor zur Verfügung, der auf dem GNU-EMACS-Editor (vgl. [Sta86]) aufsetzt und auch die Steuerung anderer

[7] unterstützt durch die Digital Equipment Corporation unter Vertrag EERP/GY–037

Spezifikations- und Verifikationswerkzeuge von der Editorebene aus vorsieht [Weg94]. Ein weiteres Werkzeug ermöglicht die Verwaltung von tTLA$^+$-Spezifikationen, die sich aus einzelnen Modulen zusammensetzen [Gei93, Mei94]. Dieser Browser analysiert Spezifikationen und extrahiert daraus die Topologie von tTLA$^+$-Modulen, die bei Spezifikationen realistischer Systeme komplex werden kann. Das Werkzeug besitzt eine graphische Benutzerschnittstelle, auf der die Modultopologie angezeigt wird. Der Benutzer kann im Stil von Hypertext relevante Teile des Spezifikationstextes aufrufen oder sich zusätzliche textuelle Erläuterungen anzeigen lassen. Weiterhin existiert ein Werkzeug zur Vereinfachung von Spezifikationen anhand einfacher Termersetzungsregeln [Raf93].

Ausführbare Spezifikationen, die nur eine endliche Anzahl an Zuständen erreichen und nur über transitionsartige Aktionen verfügen, können interpretiert werden [Lüb91, Gra93]. Ein Interpreter übersetzt Variablen und Aktionen in den Code der Programmiersprache C (vgl. [KR90]) und bindet diesen mit einer Ablaufumgebung zu einem Simulator. Die Zustandsübergänge werden graphisch dargestellt. Die Auswahl der Aktionen erfolgt entweder benutzergeführt oder zufällig. Ein Zustandsgraph aller durchlaufenen Zustände kann generiert und visualisiert werden. Alternativ zur Interpretation können spezifizierte Systeme auch bereits in einem frühen Entwicklungsstadium in einen realen Prototyp überführt werden. Innerhalb einer studentischen Projektgruppe wurde ein Prototypgenerator entworfen und prototypisch implementiert [Pro93].

Audiovisuelle Animation von Spezifikationen erleichtert das Verständnis von Spezifikationen besser als einfache Interpretation. Das Animationswerkzeug VisTLA [Kus93] erlaubt die Erstellung graphischer und akustischer Animations-Szenarien mit Hilfe des Animationsgenerators Zeus [Bro92]. Das Verhalten von Spezifikationen kann interpretiert und audiovisuell animiert werden. Eine Reihe typischer Animations-Szenarien für verteilte Systeme wurde im Rahmen einer studentischen Projektgruppe entworfen [Pro94].

3.5.2 Unterstützung der Verifikation

Wie bereits erwähnt, kann man Beweise von TLA-Formeln entweder durch logische Deduktion (siehe Abschnitt 3.3.3) oder durch zustandsbasierte Methoden (siehe Abschnitt 3.3.4) führen.

Auf der Basis des auf Termersetzung basierenden Theorembeweisers Larch [GGH90] haben Engberg et al. [EGL92] ein Werkzeug TLP entwickelt, das Spezifikationen und vollständige Beweisskripten als Eingabe hat. Die in den Beweisskripten aufgeführten Lemmata werden von Larch bewiesen. Dadurch wird der Deduktionsbeweis nicht vereinfacht, sondern nur seine Korrektheit abgesichert. Um auch die Erstellung von Beweisskripten zu unterstützen, wurde in der Dortmunder Arbeitsgruppe der Lemmagenerator TOAST entwickelt [EHK93, Eic92, Bla94]. Wie in Abschnitt 3.3.3 gezeigt wurde, erfolgt die Zerlegung eines Beweises nach einem festen Schema. TOAST nutzt dieses Schema aus. Der Benutzer muß außer den Spezifikationen von Fein- und Grobsystem nur die Zustandsabbildung, eine Invariante und bei Fairnessbeweisen Folgen an Zustandsprädikaten angeben. TOAST generiert daraus eine Reihe von Lemmata in der Eingabesprache des Theorembeweisers OTTER [McC94]. Außerdem wird ein Steuerskript erzeugt, anhand dessen OTTER die Lemmata beweist. OTTER basiert auf der Hyperresolution und erlaubt den vollautomatischen Beweis eines Lemmas, vor dessen Ablauf allerdings eine Klauselauswahlstrategie angegeben werden muß. Tests von TOAST und OTTER haben gezeigt, daß bei einer geeigneten Beschreibung von Systemspezifikationen und Klauselauswahlstrategien ein hoher Anteil der von TOAST erzeugten Lemmata

von OTTER direkt bewiesen werden kann. Bei der Verifikation der Safetyeigenschaften einer Spezifikation des Schiebefensterprotokolls nach [Tan89] konnten 121 von 128 durch TOAST erzeugte Lemmata direkt bewiesen werden. Die Zerlegung nicht direkt beweisbarer Lemmata in Sublemmata ist aufgrund der sehr rigiden Eingabesprache von OTTER kompliziert. Das Werkzeug wurde deshalb erweitert [Gei94]. TOAST gibt die von ihm erzeugten Lemmata nicht mehr direkt in der Syntax von OTTER sondern in einer Spezifikationssprache aus, die den Beweisskripten von TLP ähnlich ist. Ein solches Beweisskript kann manuell modifiziert werden, wobei auch die Klauselauswahlstrategie von OTTER für einzelne Beweise geändert werden kann. Das Beweisskript wird dann in die OTTER-Eingabesyntax übersetzt.

Methoden, die auf Zustandsraumanalyse basieren, können Beweise im Gegensatz zu Theorembeweisern vollautomatisch führen. Allerdings sind die so beweisbaren Spezifikationen eingeschränkt, da sie nur eine endliche Anzahl an erreichbaren Zuständen modellieren dürfen. Die Aktionen müssen so beschaffen sein, daß man den Folgezustand effizient aus dem Zustand vor der Ausführung der Aktion berechnen kann. Ein Refinement Mapping-Suchwerkzeug [Sch91, Pot95] erspart die manchmal komplizierte Suche nach Refinement Mappings. Für die Fein- und die Grobspezifikation werden die Erreichbarkeitsgraphen aufgestellt. Durch Backtracking wird nun eine Abbildung von den Zuständen des Feinsystems zu denen des Grobsystems gesucht, die die Refinement Mapping-Eigenschaften **R1**, **R2** und **R3** erfüllt. Das Werkzeug führt keine Livenessbeweise. Durch Angabe partieller Refinement Mappings kann der Suchraum des Backtrackings eingeengt werden.

In einem alternativen Ansatz [Her93, HKKS94, SK93, Wie94] wird Model Checking zum Beweis von Refinement Mappings verwendet. Um das bei realistischen Systemen auftretende Problem der Zustandsexplosion abzufangen, beschränkt das Werkzeug RMCHECK die zu einem Zeitpunkt zu speichernden Zustände des Erreichbarkeitsgraphen („On-the-fly“-Model Checking, siehe Abschnitt 3.1.5). Es erhält als Eingabe Spezifikationen des Fein- und Grobsystems sowie eine Zustandsabbildung, die es auf die Refinement Mapping-Eigenschaften untersucht. Die Eigenschaften **R1**, **R2** und **R3** können durch Betrachtung einzelner Zustände oder Zustandspaare überprüft werden. Die schon untersuchten Zustände werden nach dem Supertrace-Verfahren [Hol88] ökonomisch gespeichert. Aufgrund der Einschränkung der Beschreibung von Liveness auf schwache und starke Fairness kann man zeigen, daß bei der Untersuchung der Eigenschaft **R4** fast immer nur die Zustände eines Teils des Erreichbarkeitsgraphen zu einem Zeitpunkt gespeichert werden müssen (vgl. [HKKS94]). Soll zum Beispiel die Garantie der schwachen Fairness einer Aktion λ des Grobsystems durch das Feinsystem nachgewiesen werden, reicht es aus, zu jedem Zeitpunkt immer nur genau eine starke Zusammenhangskomponente derjenigen Zustände s des Feinsystems zu untersuchen, in deren Abbildungen $f(s)$ die Aktion λ schaltbar ist. Es muß dabei überprüft werden, ob durch die Fairnessanforderungen des Feinsystems das Verlassen der starken Zusammenhangskomponente erzwungen wird.

Kapitel 4

Kompositionale Spezifikation und strukturierte Verifikation

Wir stellen einen auf TLA basierenden kompositionalen Spezifikationsstil vor, der die in den Abschnitten 2.5 und 3.2.3 gestellten Forderungen nach Unterstützung der Modularität von Systemen und nach einfacher Anwendbarkeit erfüllt. Der Ansatz wurde bereits in [HK94] veröffentlicht.

Einzelne Ressourcen oder Eigenschaften (Constraints) eines Systems werden als eigenständige modulare Bausteine, sogenannte Prozesse, beschrieben und durch eine TLA-Formel in kanonischer Form (siehe Abschnitt 3.3.2) spezifiziert. Die Komposition der Prozesse zu einem System erfolgt wie bei der Kompositionsmethode von Abadi und Lamport [AL93a, AL93b] durch logische Konjunktion. Die TLA-Formeln der Prozesse werden zu einer TLA-Formel, die das Gesamtsystem spezifiziert, konjungiert. Im Gegensatz zu Abadi und Lamport wird die Interaktion zwischen den Prozessen nicht durch gemeinsam genutzte Variablen beschrieben. Stattdessen werden Aktionen einzelner Prozesse so miteinander gekoppelt, daß sie nur gleichzeitig schalten dürfen. Wie in Abschnitt 3.1.4 ausgeführt wurde, eignet sich diese synchrone Kopplungsmethode gut, Systeme nicht nur ressourcenorientiert, sondern auch constraintorientiert (vgl. [VSvS88]) in Prozesse zu zerlegen.

Da in unserem Spezifikationsstil die Komposition der logischen Konjunktion von TLA-Formeln der Prozesse zu der TLA-Formel des Systems entspricht, wird jede Eigenschaft, die von einem Prozeß erfüllt wird, zugleich auch vom System erfüllt. Dazu müssen die Prozesse allerdings zueinander widerspruchsfrei sein, so daß die Systemformel erfüllbar ist. Diese spezielle Form der Kompositionalität erlaubt die Zerlegung von TLA-Verfeinerungsbeweisen (siehe Abschnitt 3.3.3) in eine Folge wesentlich einfacherer Teilbeweise. Wenn sowohl das detailliertere System Φ als auch das abstraktere System Ψ kompositional aufgebaut sind, kann man für jeden Prozeß von Ψ separat nachweisen, daß er von Φ erbracht wird. In der Regel muß bei einem derartigen Teilbeweis nicht das gesamte detailliertere System Φ berücksichtigt werden, sondern nur ein weniger komplexes Subsystem von Φ. Außerdem muß noch bewiesen werden, daß die Kopplung der Prozeßaktionen in Φ zu der Kopplung der Prozeßaktionen in Ψ konsistent ist. Dieser Beweis profitiert sehr von den Teilbeweisen und ist im Normalfall trivial.

Die Widerspruchsfreiheit von miteinander gekoppelten Prozessen wird durch spezielle Stilkonventionen sichergestellt. Durch die Stilkonventionen werden einige Restriktionen an den

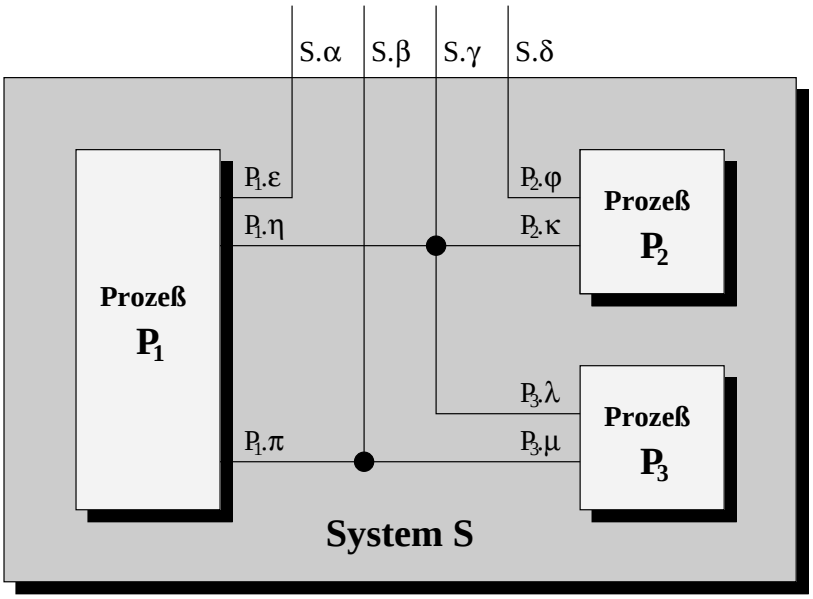

Abbildung 4.1: Systemkomposition durch gekoppelte Aktionen

Aufbau der Prozeßspezifikationen und der Kopplungsbeschreibungen gestellt, ohne jedoch die Mächtigkeit des Spezifikationsstils einzuschränken. Außerdem gewährleisten die Stilkonventionen, daß man TLA-Systemformeln in kanonische Prozeßformeln überführen kann. Dadurch werden hierarchische Spezifikationen möglich, in denen aus Prozessen komponierte Systeme als Prozesse umfassenderer Systeme dienen.

Die praktische Anwendung des Spezifikationsstils wird durch die Spezifikationssprache cTLA+ [Hey95] erleichtert. Sie erlaubt, Prozeßtypen zu spezifizieren. Durch generische Prozeßtypparameter kann man mit einer Spezifikation eine Klasse von Prozeßtypen beschreiben. Außerdem besitzt cTLA+ Operationen, mit denen man unter anderem die Instantiierung von Prozeßtypen und die Komposition von Prozessen zu Systemen spezifizieren kann.

Der Aufbau dieses Kapitels gliedert sich folgendermaßen: Zuerst wird in Abschnitt 4.1 der Spezifikationsstil vorgestellt. Anschließend erläutern wir in Abschnitt 4.2, wie man Verfeinerungsbeweise in Beweise einzelner Komponenten zerlegt. In Abschnitt 4.3 werden die Spezifikationssprache cTLA$^+$ sowie Methoden, mit denen der Einsatz des Stils für reale Systeme vereinfacht werden kann, eingeführt.

4.1 Der kompositionale Spezifikationsstil

In Abbildung 4.1 wird als Beispiel ein System S skizziert, das aus den Einzelprozessen P_1, P_2 und P_3 zusammengesetzt ist. Die Prozesse werden durch TLA-Spezifikationen modelliert. Dabei wird das Verhalten von P_1 durch die Aktionen ε, η und π, von P_2 durch φ und κ und von P_3 durch λ und μ beschrieben. Die Interaktion zwischen den Prozessen wird durch gekoppelte Aktionen modelliert. Die Aktion π in Prozeß P_1 und die Aktion μ in Prozeß P_3 dürfen nur zueinander simultan ausgeführt werden. Das gleiche gilt für die Aktionen η aus

P_1, κ aus P_2 und λ aus P_3. Das Kopplungsmodell sieht außerdem Aktionen vor, die nicht mit Aktionen aus anderen Prozessen gekoppelt sind (ε in P_1, φ in P_2). Die Aktionen des Gesamtsystems S sind zum einen die gekoppelten Aktionen, die wir als β und γ bezeichen, sowie die prozeßlokalen Aktionen, die im System α und δ genannt werden. Im Gegensatz zu CCS [Mil80] und LOTOS [ISO89b] sieht unser Spezifikationsstil keine Verdeckung systeminterner Aktionen vor. In einem umfassenderen System T, das S als Prozeß enthält, können α, β, γ und δ mit Aktionen anderer Prozesse aus T gekoppelt werden.

Der Datentransfer zwischen zwei Prozessen wird durch Aktionsparameter modelliert. Zwei Aktionen dürfen gleichzeitig nur ausgeführt werden, wenn Parameter mit identischen Bezeichnern den gleichen Wert besitzen. So werden im Beispiel die Aktionen $P_1.\pi(p)$ und $P_3.\mu(p,q)$ nur ausgeführt, wenn p in beiden Aktionen denselben Wert hat.

Im folgenden werden zunächst das Prozeßmodell (Abschnitt 4.1.1) und das Kopplungsmodell (Abschnitt 4.1.2) des Spezifikationsstils vorgestellt. Anschließend erläutern wir in Abschnitt 4.1.3 die Stilkonventionen. In Abschnitt 4.1.4 wird nachgewiesen, daß bei Einhaltung der Stilkonventionen komponierte Systeme widerspruchsfrei und in monolithische Prozeßspezifikationen überführbar sind.

4.1.1 Prozeßmodell

Zur Modellierung einzelner Prozesse verwenden wir TLA-Formeln in kanonischer Form (siehe Abschnitt 3.3.2), deren Fairnessannahmen jedoch abgeändert sind:

$$P \triangleq \left\{ \begin{array}{l} P.Init \wedge \\ \Box[\exists p \in P.ptyp_{\pi_1} : P.\pi_1(p) \vee \ldots \vee \exists p \in P.ptyp_{\pi_n} : P.\pi_n(p)]_{P.V} \wedge \\ \forall p \in P.ptyp_{\pi_i} : WF_{P.V}(P.e\pi_i(p)) \wedge \ldots \wedge \forall p \in P.ptyp_{\pi_j} : WF_{P.V}(P.e\pi_j(p)) \wedge \\ \forall p \in P.ptyp_{\pi_k} : SF_{P.V}(P.e\pi_k(p)) \wedge \ldots \wedge \forall p \in P.ptyp_{\pi_l} : SF_{P.V}(P.e\pi_l(p)) \end{array} \right.$$

Ein Prozeß P besitzt einen Satz privater Variablen $P.V$, die nicht von anderen Prozessen gelesen oder verändert werden dürfen. Die Safetyeigenschaften werden wie in den kanonischen TLA-Formeln (siehe Abschnitt 3.3.2) durch eine Initialbedingung $P.Init$ und eine Zustandsübergangsrelation modelliert, die aus einer Disjunktion von Aktionen besteht und Stotterschritte zuläßt. Da jede Aktion π mit jeder Parameterbelegung aus der Menge $P.ptyp_\pi$ schaltbar ist, werden die Aktionsparameter durch Existenzquantoren gebunden. Die Beschränkung auf private Variablen garantiert die Erfüllbarkeit aus Prozessen gekoppelter Systeme in Hinsicht auf die Safetyeigenschaften (siehe Abschnitt 4.1.4) und schließt Interferenzen zwischen den Variablen (vgl. [OG76, Hoa85]) aus.

Die Konsistenz der Livenesseigenschaften wird bei „Rendez-Vous“-Kopplungen im allgemeinen jedoch nicht garantiert (vgl. [Bri90]). Wir lösen dieses Problem durch Verwendung schwächerer Fairnessanforderungen an die Prozeßaktionen, die auch bei einem destruktiven Verhalten der Prozeßumgebung immer erfüllt werden. Im Gegensatz zu kanonischen TLA-Formeln beziehen sich Fairnessanforderungen im Spezifikationsstil nicht auf die Aktionen π aus der Zustandsfolgerelation, sondern immer auf Aktionen $e\pi$, die eine zusätzliche Bedingung enthalten:

$$e\pi(p : ptyp_\pi) \triangleq \pi(p) \wedge p \in e_\pi$$

Dabei wird $e\pi$ als bedingte Aktion (Conditioned Action) bezeichnet. Sie ist nur schaltbar,

wenn ihr Aktionsparameter p vom Datentyp $ptyp_\pi$ Element einer Mengenvariablen e_π ist[1]. Diese sogenannte Umgebungsbereitmengenvariable (Environment Readiness Variable, ERV) wird von der Prozeßumgebung beschrieben. Dadurch ist dem Prozeß P bekannt, für welche Parameterbelegungen eine Aktion π, die mit Aktionen aus der Umgebung gekoppelt ist, geschaltet werden kann. Bei korrekter Kopplung[2] enthält e_π nur Parameterbelegungen, für die die mit π gekoppelten Aktionen aus anderen Prozessen geschaltet werden können. Dadurch ist die Fairnessannahme der von e_π abhängigen bedingten Aktion $e\pi$ immer erfüllt (siehe Abschnitt 4.1.4). Die ERVs modellieren keine eigenständigen Systemressourcen, sondern sind Hilfsvariablen, die durch Zustandsfunktionen über Variablen der Partnerprozesse definiert sind. Da sie von Prozeß P nur gelesen werden und nur von dessen Umgebungsprozessen verändert werden, treten keine Interferenzen auf (vgl. [OG76]).

Die Prozesse des Spezifikationsstils sind nach dieser Definition offene Subsysteme, die in eine Systemumgebung eingebettet sind, die sie selbst beeinflussen und von der sie beeinflußt werden. Wie in CCS oder LOTOS können sie jedoch auch als eigenständige Einheiten betrachtet werden. In diesem Fall wird der Prozeß durch Kopplung mit einer universellen Systemumgebung, die alle vom Prozeß modellierten Eigenschaften toleriert, zu einem geschlossenen System komponiert, in dem die Aktionen nicht mit Aktionen aus Umgebungsprozessen gekoppelt sind und in dem die Fairnessannahmen nicht durch die ERVs eingeschränkt werden:

$$S_g \triangleq P \wedge \Box(P.e\pi_1(p) = P.ptyp_{\pi_1} \wedge \ldots \wedge P.e\pi_n(p) = P.ptyp_{\pi_n}) \tag{4.1}$$

P wird nicht durch seine Umgebung beeinflußt, da seine Aktionen $\pi_i(p)$ nicht mit Aktionen aus anderen Prozessen gekoppelt sind. Die Fairness der Aktionen wird nicht von seiner Umgebung eingeschränkt ($P.e\pi_i(p) = P.ptyp_{\pi_i}$). Deshalb ist jede Prozeßaktion π_i von P zu ihrer bedingten Aktion $e\pi_i$ äquivalent und S_g läßt sich in eine kanonische TLA-Formel umformen.

4.1.2 Prozeßkopplung

Systeme S, die durch Komposition von Prozessen gebildet sind, werden durch Konjunktion der Prozeßformeln P_i (siehe Abschnitt 4.1.1) und einer zusätzlichen Kopplungsbedingung CC definiert:

$$S \triangleq P_1 \wedge P_2 \wedge \ldots \wedge P_m \wedge CC$$

CC wird als Kopplungsbeschreibung bezeichnet. Durch sie werden die Kopplung der Prozeßaktionen und die Definition der Umgebungsbereitmengenvariablen (ERVs) festgelegt. Wir werden in Abschnitt 4.1.4 zeigen, daß unter Voraussetzung einiger Stilkonventionen kompositionale Systemformeln in kanonische Formeln

$$S \triangleq \left\{ \begin{array}{l} S.Init \wedge \\ \Box[\exists p \in S.ptyp_{\pi_1} : S.\pi_1(p) \vee \ldots \vee \exists p \in S.ptyp_{\pi_n} : S.\pi_n(p)]_{S.V} \wedge \\ \forall p \in S.ptyp_{\pi_i} : WF_{S.V}(S.e\pi_i(p)) \wedge \ldots \wedge \forall p \in S.ptyp_{\pi_j} : WF_{S.V}(S.e\pi_j(p)) \wedge \\ \forall p \in S.ptyp_{\pi_k} : SF_{S.V}(S.e\pi_k(p)) \wedge \ldots \wedge \forall p \in S.ptyp_{\pi_l} : SF_{S.V}(S.e\pi_l(p)) \end{array} \right.$$

[1] O.B.d.A. nehmen wir an, daß alle Aktionen mindestens einen Parameter besitzen. Parameterlosen Aktionen kann man einen Aktionsparameter hinzufügen, der keine Bedeutung für die Interaktion zwischen Prozessen hat.

[2] Die Korrektheit von Kopplungen wird durch Stilkonventionen (siehe Abschnitt 4.1.3) sichergestellt.

überführt werden können. $S.Init$ steht für die Initialbedingung und entspricht der Konjunktion der Initialbedingungen der miteinander komponierten Prozesse. Die Parameter der Systemaktionen $\exists p \in S.ptyp_{\pi_i} : S.\pi_i(p)$ dürfen Werte aus der Menge $S.ptyp_{\pi_i}$ annehmen. Die Systemaktionen sind miteinander gekoppelte Prozeßaktionen oder prozeßinterne Aktionen. Die Fairnessanforderungen der Prozeßaktionen gehen auf die Systemaktionen über.

Der Spezifikationsstil soll dem Systementwickler ermöglichen, die Nebenläufigkeit von Prozessen sowohl durch Parallelität der Prozesse als auch durch Interleaving oder durch Mischformen zu beschreiben[3]. Zur Darstellung der Kopplungsbeschreibung CC wählen wir einen Aufbau, der für jeden der miteinander komponierten Prozesse P_i eine separate Beschreibung der lokalen Prozeßkopplungsbedingungen erlaubt. CC ist eine Konjunktion von Prozeßkopplungsbeschreibungen P_iC, in denen die für den Prozeß P_i relevante Kopplungsinformation enthalten ist:

$$CC \triangleq \Box(P_1C \wedge \ldots \wedge P_mC) \tag{4.2}$$

Die Kopplung der Prozeßaktionen von Prozeß P_i wird in P_iC durch die Aktionskopplungsbeschreibung P_iCON (P_i-Connect) beschrieben. Die ERV-Definitionsbeschreibung P_iRED (P_i-Ready) enthält die Definition der Umgebungsbereitmengenvariablen aller Aktionen von Prozeß P_i:

$$P_iC \triangleq P_iCON \wedge P_iRED \tag{4.3}$$

P_iCON ist eine Disjunktion, die für jede Prozeßaktion π_j von Prozeß P_i angibt, mit welchen Aktionen anderer Prozesse π_j gekoppelt ist. Außerdem ist in P_iCON festgelegt, aus welchen Prozessen Aktionen parallel zu π_j geschaltet werden dürfen:

$$P_iCON \triangleq \left\{ \begin{array}{l} [\,(\exists p \in P_i.ptyp_{\pi_1} : (P_i.\pi_1(p) \wedge P_i.\pi_1PA(p)) \wedge P_i.\pi_1SR) \vee \\ \quad \ldots \vee \\ \quad (\exists p \in P_i.ptyp_{\pi_n} : (P_i.\pi_n(p) \wedge P_i.\pi_nPA(p)) \wedge P_i.\pi_nSR)]_{P_i.V} \end{array} \right. \tag{4.4}$$

P_iCON toleriert alle Zustandsübergänge des Gesamtsystems, in denen die lokalen Variablen $P_i.V$ von Prozeß P_i stottern. Dadurch ist sichergestellt, daß gekoppelte Systeme gegenüber Stotterschritten invariant sind (siehe Abschnitt 3.3.2). Das Prädikat $P_i.\pi_jPA(p)$ ($P_i.\pi_j$-Peer Actions) ist eine Konjunktion aller mit $P_i.\pi_j$ gekoppelten Aktionen:

$$P_i.\pi_jPA(p) \triangleq P_k.\pi_o(p) \wedge \ldots \wedge P_l.\pi_q(p) \tag{4.5}$$

Dabei wird hier o.B.d.A. angenommen, daß die Aktionsparameter aller Prozesse mit dem gleichen Bezeichner denselben Typ haben. Für Aktionen $P_i.\varphi_j$, die nicht mit Aktionen aus anderen Prozessen gekoppelt sind, gilt

$$P_i.\varphi_jPA(p) \triangleq True \tag{4.6}$$

Die Art der Nebenläufigkeit zu Prozessen, aus denen keine Aktion mit $P_i.\pi_j$ gekoppelt ist, wird durch das Prädikat $P_i.\pi_jSR$ ($P_i.\pi_j$-System Concurrency Restriction) festgelegt. $P_i.\pi_jSR$ setzt sich aus einer Konjunktion von *Unchanged*-Bedingungen von Prozeßvariablen $P_r.V$ zusammen:

$$P_i.\pi_jSR \triangleq Unchanged(P_r.V) \wedge \ldots \wedge Unchanged(P_s.V) \tag{4.7}$$

[3] Allerdings schränken die in Abschnitt 4.1.3 beschriebenen Stilkonventionen diese Freiheit teilweise wieder ein.

Vollständiges Interleaving wird spezifiziert, indem in allen Prädikaten $P_i.\pi_j SR$ immer die Unchanged-Bedingungen aller Prozesse aufgeführt werden, die in $P_i.\pi_j PA$ keine mit $P_i.\pi_j$ gekoppelten „Peer Actions“ besitzen. Durch $\forall i,j : P_i.\pi_j SR \triangleq True$ wird Parallelität modelliert. Die Anzahl an Aktionen in der kanonischen Systemformel von S ist geringer, wenn Nebenläufigkeit zwischen den Prozessen P_i durch Interleaving beschrieben wird.

Die Belegung der Umgebungsbereitmengenvariablen (ERV) wird durch die ERV-Definitionsbeschreibung $P_i RED$ bestimmt, die aus einer Konjunktion von Prädikaten $P_i.\pi_j EQ$ ($P_i.\pi_j$-ERV-Equation) besteht:

$$P_i RED \triangleq P_i.\pi_1 EQ \wedge \ldots \wedge P_i.\pi_m EQ \tag{4.8}$$

$P_i.\pi_j EQ$ definiert die Umgebungsbereitmengenvariable (ERV) $P_i.e_{\pi_j}$:

$$P_i.\pi_j EQ \triangleq P_i.e_{\pi_j} = \{p | Enabled(P_k.\pi_o(p)) \wedge \ldots \wedge Enabled(P_l.\pi_q(p))\} \cap P_i.ge_{\pi_j} \tag{4.9}$$

Die ERV $P_i.e_{\pi_j}$ enthält alle Parameter p, mit denen die mit $P_i.\pi_j$ zu einer Systemaktion gekoppelten Aktionen $P_k.\pi_o(p), \ldots, P_l.\pi_q(p)$ schaltbar sind. Das sind gerade die in Prädikat $P_i.\pi_j PA(p)$ gekoppelten „Peer Actions“. Die Fairnessanforderungen von $P_i.\pi_j$ hängen jedoch nicht nur vom Verhalten der mit P_i gekoppelten Prozesse ab, sondern auch vom Verhalten der Umgebung des Gesamtsystems S, das ebenfalls offen ist. Zur Beschreibung der Systemumgebung von S führen wir eine zusätzliche globale Umgebungsbereitmengenvariable $P_i.ge_{\pi_j}$ ein, die von der Umgebung von S geschaltet wird.

Bei internen Aktionen hängt die ERV nur von der globalen ERV ab:

$$P_i.\varphi_j EQ \triangleq P_i.e_{\varphi_j} = P_i.ge_{\varphi_j} \tag{4.10}$$

Als Beispiel soll nun die Prozeßkopplungsbeschreibung für den Prozeß P_2 aus Abbildung 4.1 modelliert werden, dessen Aktion $P_2.\kappa$ mit den Aktionen $P_1.\eta$ und $P_3.\lambda$ zu der Systemaktion $S.\gamma$ gekoppelt wird Die Aktion $P_2.\varphi$ wird mit keiner Prozeßaktion gekoppelt und modelliert die Systemaktion $S.\delta$. Sie soll zu Prozeß P_3 in Interleaving ausgeführt werden, darf aber zu Prozeß P_1 parallel geschaltet werden. Damit dürfen $P_2.\varphi$ und $P_1.\varepsilon$ bzw. die Systemaktionen $S.\alpha$ und $S.\delta$ zueinander parallel ausgeführt werden. Aufgrund des Interleaving zu Prozeß P_3 dürfen die Systemaktionen $S.\delta$ und $S.\beta$ hingegen nicht gleichzeitg geschaltet werden. Die Belegung der Prädikate wird im folgenden gezeigt:

$$\begin{array}{ll}
P_2C & \triangleq P_2CON \wedge P_2RED \\
P_2CON & \triangleq \left\{ \begin{array}{l} [\,\exists p \in P_2.ptyp_\varphi : (P_2.\varphi(p) \wedge P_2.\varphi PA(p)) \wedge P_2.\varphi SR) \vee \\ \quad \exists p \in P_2.ptyp_\kappa : (P_2.\kappa(p) \wedge P_2.\kappa PA(p)) \wedge P_2.\kappa SR)]_{P_2.V} \end{array} \right. \\
P_2.\varphi PA(p) & \triangleq True \\
P_2.\varphi SR & \triangleq Unchanged(P_3.V) \\
P_2.\kappa PA(p) & \triangleq P_1.\eta(p) \wedge P_3.\lambda(p) \\
P_2.\kappa SR & \triangleq True \\
P_2RED & \triangleq P_2.\varphi EQ \wedge P_2.\kappa EQ \\
P_2.\varphi EQ & \triangleq P_2.e_\varphi = P_2.ge_\varphi \\
P_2.\kappa EQ & \triangleq P_2.e_\kappa = \{p | Enabled(P_1.\eta(p)) \wedge Enabled(P_3.\lambda(p))\} \cap P_2.ge_\kappa
\end{array}$$

4.1.3 Stilkonventionen

Das im vorhergehenden Abschnitt definierte Kopplungsprinzip bietet dem Spezifizierer ein hohes Maß an Freizügigkeit bei der Entwicklung von Systemspezifikationen. Man kann jedoch leicht zeigen, daß man damit auch unerfüllbare und nicht in monolithische Spezifikationen überführbare Systemspezifikationen erzeugen kann. Um das auszuschließen, führen wir einige Stilkonventionen ein, die als Basis für die Verifikationen im folgenden Abschnitt dienen. Durch diese Stilkonventionen werden einige Anforderungen an die Prozeßspezifikationen und an die Kopplungsbeschreibung CC gestellt. Die Mächtigkeit des kompositionalen Spezifikationsstils wird durch diese Stilkonvention nur soweit vermindert, daß die Spezifikation realer Systeme nicht behindert wird.

Wir führen zunächst zwei Stilkonventionen P_1 und P_2 ein, die die Verwendung von Prozeßaktionen einschränken:

P_1: In einem Prozeß P ist eine Aktion $P.\pi_i$, für die eine faire bedingte Aktion $P.e\pi_i$ existiert, zu allen anderen Aktionen $P.\pi_j$ des Prozesses P disjunkt:

$$\forall i,j,k \in Nat : \langle s_0, s_1, \ldots\rangle [\![WF_{P.V}(P.\pi_i)]\!] \wedge s_k [\![P.\pi_i]\!] s_{k+1} \wedge i \neq j \Rightarrow \neg s_k [\![P.\pi_j]\!] s_{k+1}$$
$$\forall i,j,k \in Nat : \langle s_0, s_1, \ldots\rangle [\![SF_{P.V}(P.\pi_i)]\!] \wedge s_k [\![P.\pi_i]\!] s_{k+1} \wedge i \neq j \Rightarrow \neg s_k [\![P.\pi_j]\!] s_{k+1}$$

P_2: In einem Prozeß P ist eine Aktion $P.\pi_i$, für die eine faire bedingte Aktion $P.e\pi_i$ existiert, disjunkt zu Stotterschritten:

$$\forall i,k \in Nat : \langle s_0, s_1, \ldots\rangle [\![WF_{P.V}(P.\pi_i)]\!] \wedge s_k [\![P.\pi_i]\!] s_{k+1} \Rightarrow s_k \neq s_{k+1}$$
$$\forall i,k \in Nat : \langle s_0, s_1, \ldots\rangle [\![SF_{P.V}(P.\pi_i)]\!] \wedge s_k [\![P.\pi_i]\!] s_{k+1} \Rightarrow s_k \neq s_{k+1}$$

Fünf Stilkonventionen schränken die Aktionskopplungsbeschreibung P_iCON in der Kopplungsbeschreibung CC ein:

C_1: Eine Aktion $P_k.\pi_i$ aus Prozeß P_k darf nicht mit einer anderen Aktion $P_k.\pi_j$ aus P_k gekoppelt sein. Das Peer Action-Prädikat $P_k.\pi_i PA$ (siehe Formel 4.5) hat also niemals eine Aktion $P_k.\pi_j$ als Konjunkt. P_k ist ebenfalls nicht im System Concurrency Restriction Prädikat $P_k.\pi_i SR$ als *Unchanged*-Bedingung enthalten (siehe Formel 4.7).

C_2: Jede Aktion $P_k.\pi_i$ aus Prozeß P_k ist in mindestens einem Disjunkt der Aktionskopplungsbeschreibung P_kCON enthalten (siehe Formel 4.4):

$$\exists p \in P_k.ptyp_{\pi_i} : (P_k.\pi_i(p) \wedge P_k.\pi_i PA(p)) \wedge P_k.\pi_i SR$$

C_3: Jede Aktion $P_k.\pi_i$ aus Prozeß P_k, für die eine faire bedingte Aktion $P_k.e\pi_i$ existiert, ist in höchstens einem Disjunkt der Prozeßkopplungsbeschreibung P_kCON enthalten.

C_4: Zu jedem Disjunkt der Kopplungsbeschreibung P_kCON des Prozesses P_k, in dessen Prädikat $P_k.\pi_i PA(p)$ die Aktion $P_l.\pi_j$ enthalten ist, existiert ein äquivalentes Disjunkt in der Prozeßkopplungsbeschreibung P_lCON. Wenn $P_k.\pi_i PA(p)$ die Form

$$P_k.\pi_i PA(p) \triangleq P_l.\pi_j(p) \wedge OtherActions(p)$$

hat, besitzt $P_l.\pi_j PA(p)$ die Form

$$P_l.\pi_j PA(p) \triangleq P_k.\pi_i(p) \wedge OtherActions(p)$$

Außerdem sind $P_k.\pi_i SR$ und $P_l.\pi_j SR$ zueinander gleich.

C_5: Eine Prozeßaktion $P_k.\pi_i$, für die eine faire bedingte Aktion $P_k.e\pi_i$ existiert, ist immer in einer Systemaktion gekoppelt, die in Interleaving zu allen anderen Systemaktionen steht. Das heißt, daß alle in System S gekoppelten Prozesse P_l (außer P_k selbst) in $P_k.\pi_i PA$ oder in $P_k.\pi_i SR$ aufgeführt sind.

Durch die Stilkonvention C_1 wird sichergestellt, daß niemals zwei Aktionen aus dem gleichen Prozeß miteinander gekoppelt sind. Die Konvention C_2 garantiert, daß jede Prozeßaktion in einer Systemaktion gekoppelt ist. C_3 stellt sicher, daß faire Prozeßaktionen nur in einer einzigen Systemaktion gekoppelt sein können. C_4 garantiert die Konsistenz der Prozeßkopplungsbeschreibungen. C_5 stellt sicher, daß faire Systemaktionen zueinander immer in Interleaving ausgeführt werden. Die Bedingungen sind sowohl für den Nachweis der Erfüllbarkeit als auch der Umformbarkeit kompositionaler Spezifikationen in monolithische Darstellungen (siehe Abschnitt 4.1.4) notwendig.

Außerdem benötigen wir auch die beiden folgenden Stilkonventionen, die die Definition $P_i RED$ der Umgebungsbereitmengenvariablen einschränken:

E_1: Die ERV $P_k.e_{\pi_i}$ im ERV Ready-Prädikat $P_k.\pi_i EQ$ (siehe Formel 4.9) enthält die Menge aller Parameter p, für die die Konjunktion der in $P_k.\pi_i PA$ aufgeführten Aktionen schaltbar ist[4].

E_2: Die globalen ERVs $P_k.ge_{\pi_i}$ und $P_l.ge_{\pi_j}$ sind zueinander gleich, wenn $P_k.\pi_i$ und $P_l.\pi_j$ miteinander gekoppelt sind:

$$\Box(P_k.ge_{\pi_i} = P_l.ge_{\pi_j})$$

4.1.4 Nachweis der Erfüllbarkeit komponierter Systeme

Im folgenden wird der Nachweis erbracht, daß aus Prozessen zusammengesetzte Systeme immer ein erfüllbares Verhalten besitzen, wenn die oben angegebenen Stilkonventionen eingehalten werden. Eine Systemspezifikation ist erfüllbar, wenn sie mindestens eine Zustandsfolge aus der Menge St^∞ (siehe Abschnitt 3.3.1) erfüllt. Abadi und Lamport weisen nach [AL91b], daß eine TLA-Formel immer erfüllbar ist, wenn die folgenden Bedingungen gelten:

1. Das Initialprädikat wird von mindestens einem Zustand erfüllt.
2. Die Zustandsübergangsrelation läßt Stotterschritte zu.
3. Liveness wird nur durch die Annahme starker bzw. schwacher Fairness an Aktionen modelliert.
4. Fairnessanforderungen werden nur an Aktionen gestellt, die in der Zustandsfolgerelation enthalten sind[5].

[4] Durch die Stilkonvention C_1 ist sichergestellt, daß $P_k.e_{\pi_i}$ nur von Aktionen der Umgebung des Prozesses P_k, nicht aber von Aktionen von P_k abhängt.

[5] Jede faire Aktion impliziert die Zustandsfolgerelation.

Die ersten beiden Bedingungen garantieren die Existenz mindestens einer Zustandsfolgerelation. Durch die dritte Bedingung wird sichergestellt, daß die Livenessannahmen keine zusätzlichen Safetyannahmen enthalten („Machine Closure“, siehe auch Abschnitt 3.3.4). Dadurch wird ausgeschlossen, daß die Livenessanforderungen zu den Safetyanforderungen widersprüchlich sind. Die vierte Annahme schließlich stellt sicher, daß Fairnessanforderungen nur an Aktionen gestellt werden, deren Schaltung nicht durch die Zustandsübergangsrelation verboten wird.

Wir werden nun zeigen, daß Systeme, die nach unserem Spezifikationsstil aus Prozessen komponiert sind, diese Bedingungen erfüllen, wenn sie die Stilkonventionen einhalten und außerdem jeder einzelne Prozeß selbst erfüllbar ist. Man kann ein komponiertes System S_C durch die Formel

$$S_C \Leftrightarrow \left\{ \begin{array}{l} P_1.Init \wedge \ldots \wedge P_m.Init \wedge \\ \Box[\,(P_1ZFR \wedge P_1CON \wedge P_1RED) \wedge \\ \quad \ldots \wedge \\ \quad (P_mZFR \wedge P_mCON \wedge P_mRED)]_{(P_1.V,\ldots,P_m.V)} \wedge \\ P_1.SFA \wedge \ldots \wedge P_mSFA \wedge \\ P_1.WFA \wedge \ldots \wedge P_mWFA \end{array} \right. \tag{4.11}$$

beschreiben. Dabei ist P_iZFR die Zustandsübergangsrelation von Prozeß P_i. $P_i.SFA$ und $P_i.WFA$ beschreiben die Konjunktion der starken bzw. schwachen Fairnessanforderungen der Aktionen von Prozeß P_i. Die dargestellte Zustandsfolge wurde durch Anwendung der TLA-Regel STL5 (siehe Abbildung 3.2) auf die Zustandsfolgerelationen P_iZFR und die Aktionskopplungsbeschreibungen P_iCON hergeleitet. Aufgrund der TLA-Regel INV2 wurden zu der neu entstandenen Zustandsübergangsrelation die invarianten ERV-Definionsbeschreibungen P_iRED hinzugefügt.

Da wir die vierte Bedingung nicht direkt verifizieren können, definieren wir zunächst eine neue Spezifikation $\widehat{S_C}$, die gleiche Initialprädikate und Zustandsfolgerelationen wie S_C besitzt, sich aber durch stärkere Fairnessannahmen auszeichnet:

$$\widehat{S_C} \triangleq \left\{ \begin{array}{l} P_1.Init \wedge \ldots \wedge P_m.Init \wedge \\ \Box[\,(P_1ZFR \wedge P_1CON \wedge P_1RED) \wedge \\ \quad \ldots \wedge \\ \quad (P_mZFR \wedge P_mCON \wedge P_mRED)]_{(P_1.V,\ldots,P_m.V)} \wedge \\ P_1.\widehat{SF}A \wedge \ldots \wedge P_m\widehat{SF}A \wedge \\ P_1.\widehat{WF}A \wedge \ldots \wedge P_m\widehat{WF}A \end{array} \right. \tag{4.12}$$

In $P_k.\widehat{SF}A$ und $P_k.\widehat{WF}A$ wird jede Fairnessanforderung an eine Aktion $P_k.e\pi_i(p)$ durch eine gleichartige Fairnessanforderung an die Aktion

$$P_k.\widehat{e\pi_i}(p) \triangleq P_k.e\pi_i(p) \wedge P_k.\pi_iPA(p) \wedge P_k.\pi_iSR \tag{4.13}$$

ersetzt. Als Umgebungsbereitmengenvariable verwendet $P_k.\widehat{e\pi_i}(p)$ die globale Umgebungsbereitmengenvariable (siehe Abschnitt 4.1.2) $P_k.ge_{\pi_k}$ der Aktion $P_k.e\pi_i(p)$.

Wir zeigen zunächst im folgenden Satz, daß S_C und $\widehat{S_C}$ zueinander äquivalent sind, wenn sie die im vorangegangenen Abschnitt aufgestellten Stilkonventionen einhalten. Dazu verwen-

den wir die Spezifikation S_{C_s}, die die Safetyeigenschaften von S_C und $\widehat{S_C}$ beschreibt:

$$S_{C_s} \triangleq \left\{ \begin{array}{l} P_1.Init \wedge \ldots \wedge P_m.Init \wedge \\ \Box[\,(P_1 ZFR \wedge P_1 CON \wedge P_1 RED) \wedge \\ \quad \ldots \wedge \\ \quad (P_m ZFR \wedge P_m CON \wedge P_m RED)]_{(P_1.V,\ldots,P_m.V)} \end{array} \right. \tag{4.14}$$

Satz 4.1 *Unter Einhaltung der Stilkonventionen gilt für alle stark oder schwach fairen Aktionen $P_k.e\pi_i(p)$ in den Spezifikationen S_C und $\widehat{S_C}$*

$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \Rightarrow (SF_{P.V}(P_k.e\pi_i(p)) \Leftrightarrow SF_{S_C.V}(\widehat{P_k.e\pi_i}(p)))$$
$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \Rightarrow (WF_{P.V}(P_k.e\pi_i(p)) \Leftrightarrow WF_{S_C.V}(\widehat{P_k.e\pi_i}(p)))$$

Beweis: Da die Fairness einer Aktion π durch

$$\Box\Diamond\langle\pi\rangle_V \wedge \Box\Diamond \;/\Diamond\Box Enabled(\langle\pi\rangle_V)$$

beschrieben wird (siehe Abschnitt 3.3.2), reicht es aus die Gültigkeit der folgenden Formeln zu beweisen:

$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \Rightarrow (Enabled(\langle P_k.e\pi_i(p)\rangle_{P_k.V}) \Leftrightarrow Enabled(\langle\widehat{P_k.e\pi_i}(p)\rangle_{S_C.V})) \tag{4.15}$$
$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \Rightarrow (\langle P_k.e\pi_i(p)\rangle_{P_k.V} \Leftrightarrow \langle\widehat{P_k.e\pi_i}(p)\rangle_{S_C.V}) \tag{4.16}$$

Wir verifizieren zunächst 4.15 und beginnen mit dem Nachweis von

$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \wedge Enabled(\langle P_k.e\pi_i(p)\rangle_{P_k.V}) \Rightarrow Enabled(\langle\widehat{P_k.e\pi_i}(p)\rangle_{S_C.V}) \tag{4.17}$$

Aus der Stilkonvention E_1 und aus der Definition der Kopplungsbeschreibung, nach der $Enabled(P_k.\pi_i SR) \Leftrightarrow True$ gilt, folgt die Eigenschaft

$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \wedge Enabled(P_k.e\pi_i(p)) \Rightarrow Enabled(\widehat{P_k.e\pi_i}(p))$$

Da außerdem nach der Definition der Variablen von S_C

$$Unchanged(S_C.V) \Rightarrow Unchanged(P_k.V)$$

gilt, folgt daraus die Formel 4.17. Der Beweis von

$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \wedge Enabled(\langle\widehat{P_k.e\pi_i}(p)\rangle_{S_C.V}) \Rightarrow Enabled(\langle P_k.e\pi_i(p)\rangle_{P_k.V}) \tag{4.18}$$

folgt aus der Stilkonvention P_2. Da $Enabled(\langle\widehat{P_k.e\pi_i}(p)\rangle_{S_C.V}) \Rightarrow Enabled(P_k.e\pi_i(p))$ trivialerweise gilt, folgt aus P_2 jedoch $P_k.e\pi_i(p) \Rightarrow \langle P_k.e\pi_i(p)\rangle_{P_k.V}$, so daß 4.18 wahr ist. Aus den Formeln 4.17 und 4.18 folgt die Gültigkeit der Formel 4.15.

Im nächsten Beweisschritt verifizieren wir die Eigenschaft 4.16. Der Beweis wird wiederum in zwei Schritte zerlegt. Zuerst beweisen wir

$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \wedge \langle P_k.e\pi_i(p)\rangle_{P_k.V} \Rightarrow \langle\widehat{P_k.e\pi_i}(p)\rangle_{S_C.V} \tag{4.19}$$

Da $P_k.e\pi_i(p)$ fair ist, existiert aufgrund der Stilkonventionen P_1, C_2 und C_3 im Peer Actions-Prädikat $P_k CON$ genau ein Disjunkt

$$P_k.\pi_i \wedge P_k.\pi_i PA(p) \wedge P_k.\pi_i SR$$

Wegen der Stilkonvention C_4 enthalten auch die Peer Action-Prädikate $P_l CON$ der Prozesse, aus denen eine Aktion mit $P_k.\pi_i$ gekoppelt ist, dieses Disjunkt genau einmal. Da nach Definition $\langle P_k.e\pi_i(p)\rangle_{P_k.V} \Rightarrow \langle P_k.\pi_i(p)\rangle_{P_k.V}$ gilt, folgt daraus sofort die Formel 4.19.

Die Formel

$$\forall p \in P_k.ptyp\pi_i : S_{C_s} \wedge \langle \widehat{P_k.e\pi_i(p)}\rangle_{S_C.V} \Rightarrow \langle P_k.e\pi_i(p)\rangle_{P_k.V} \tag{4.20}$$

folgt wie die Formel 4.18 sofort aus der Stilkonvention P_2.

Die Behauptung des Satzes ergibt sich aus den Formeln 4.15 und 4.16. □

Aus dem Satz 4.1 und der Definition von $\widehat{S_C}$ kann man sofort $S_C \Leftrightarrow \widehat{S_C}$ folgern. Wir beweisen nun, daß $\widehat{S_C}$ und damit auch S_C erfüllbar sind. Dazu zeigen wir die Gültigkeit der vier am Anfang des Abschnitts angegebenen Bedingungen.

1. Für jeden zu $\widehat{S_C}$ gekoppelten Prozeß P_i existiert ein Zustand s_{i_0}, der die Initialbedingung $P_i.Init$ erfüllt ($s_{i_0}[\![P_i.Init]\!]$). Das Tupel $(s_{0_0} \times \ldots \times s_{m_0})$ erfüllt dann aber die Anfangsbedingung von $\widehat{S_C}$:

   $$(s_{0_0} \times \ldots \times s_{m_0})[\![P_1.Init \wedge \ldots \wedge P_m.Init]\!]$$

   weil die Variablen der Prozesse P_i privat sind. $P_i.Init$ trifft nämlich nur Aussagen über s_{i_0}, nicht aber über die Anfangszustände s_{l_0} der anderen Prozesse P_l. Deshalb gilt auch

   $$(s_{0_0} \times \ldots \times s_{m_0})[\![P_i.Init]\!]$$

   Diese Eigenschaft trifft für alle Prozesse P_l zu, so daß aus der Abgeschlossenheit von TLA-Formeln gegenüber der Konjunktion (siehe Formel 3.10[6] in Abschnitt 3.3.1) die Annahme sofort folgt.

2. Die Zustandsfolgerelation von $\widehat{S_C}$ läßt Stotterschritte zu (siehe Formel 4.12).

3. Die Livenessanforderungen von $\widehat{S_C}$ werden nur durch starke und schwache Fairnessanforderungen an Aktionen modelliert.

4. Der Nachweis, daß jede faire Aktion die Zustandsfolgerelation impliziert, ergibt sich aus dem Beweis von Satz 4.1.

Zum Schluß zeigen wir noch, daß zu einer kompositionalen Systemspezifikation immer eine äquivalente Spezifikation S_M als kanonische Prozeßformel existiert:

$$S_M \triangleq \left\{ \begin{array}{l} S_M.Init \wedge \\ \Box[\exists p \in S_M.ptyp_{\pi_1} : S_M.\pi_1(p) \vee \ldots \vee \exists p \in S_M.ptyp_{\pi_n} : S_M.\pi_n(p)]_{S_M.V} \wedge \\ \forall p \in S_M.ptyp_{\pi_i} : WF_{S_M.V}(S_M.e\pi_i(p)) \wedge \ldots \wedge \forall p \in S_M.ptyp_{\pi_j} : \\ \quad WF_{S_M.V}(S_M.e\pi_j(p)) \wedge \\ \forall p \in S_M.ptyp_{\pi_k} : SF_{S_M.V}(S_M.e\pi_k(p)) \wedge \ldots \wedge \forall p \in S_M.ptyp_{\pi_l} : SF_{S_M.V}(S_M.e\pi_l(p)) \end{array} \right.$$

Wir verwenden die zu S_C äquivalente Spezifikation $\widehat{S_C}$ (siehe Formel 4.12). Als Initialbedingung $S_M.Init$ verwenden wir die Konjunktion der Initialbedingungen der Einzelprozesse:

$$S_M.Init \triangleq P_1.Init \wedge \ldots \wedge P_m.Init$$

[6] Aufgrund der Äquivalenz in Formel 3.8 gilt die Konjunktion auch für einzelne Zustände.

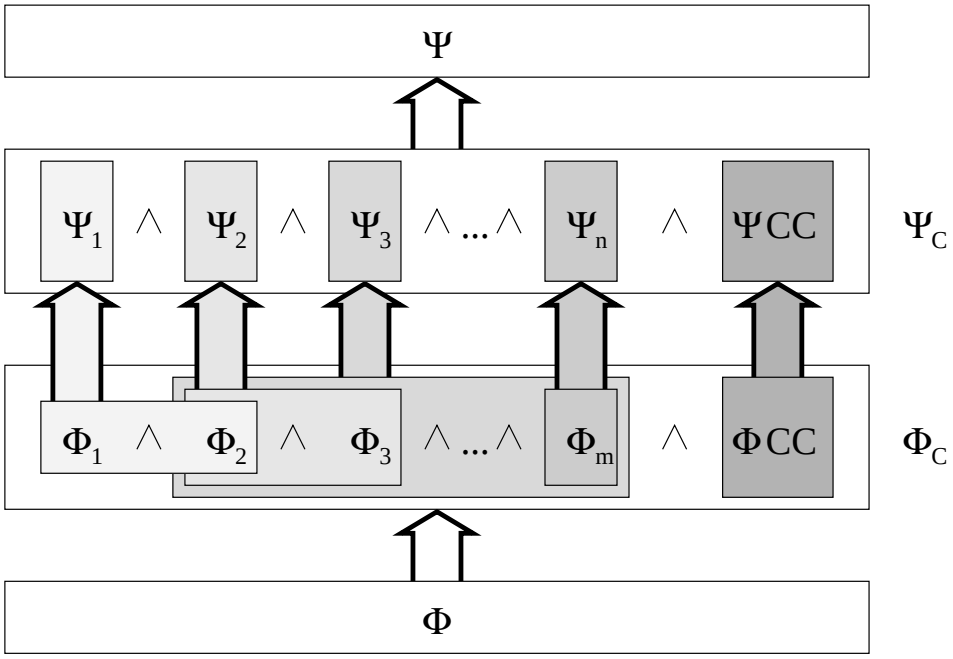

Abbildung 4.2: Strukturierte Verifikation kompositionaler Spezifikationen

Die Zustandsübergangsrelation $S_M.ZFR$ wird erzeugt, indem die Zustandsübergangsrelation von $\widehat{S_C}$ in die disjunktive Normalform überführt wird. Als Fairnessanforderung wird die Konjunktion der Fairnessannahmen in $\widehat{S_C}$ verwendet:

$$P_1.\widehat{SF}A \wedge \ldots \wedge P_m\widehat{SF}A \wedge$$
$$P_1.\widehat{WF}A \wedge \ldots \wedge P_m\widehat{WF}A$$

Durch die Stilkonventionen C_1, C_2, C_3, C_4, C_5, E_1 und E_2 wird sichergestellt, daß jede in $P_k.\widehat{SF}A$ oder $P_k.\widehat{WF}A$ als fair spezifizierte Aktion genau eine Aktion der Zustandsübergangsrelation in der disjunktiven Normalform impliziert. Die Überführbarkeit kompositionaler in monolithische Spezifikationen ermöglicht den modularen Aufbau von Spezifikationen, da kompositionale Systeme immer als Bausteine umfassenderer Systeme eingesetzt werden können.

4.2 Strukturierte Verifikation

Entscheidend für den Einsatz formaler Spezifikations- und Verifikationstechniken in der Praxis ist die Verfügbarkeit von Methoden zur Reduktion der hohen Komplexität realer Systeme. Insbesondere Verfeinerungsbeweise sind trotz inzwischen wirksamer Werkzeugunterstützung in der Realität oft nur mit einem sehr hohen zeitlichen Aufwand durchführbar. Als problematisch erweist sich dabei insbesondere die aufgrund der Systemkomplexität schwierige Suche nach passenden Invarianten, Refinement Mappings und bei Livenessbeweisen garantierten Zustandsfolgen (Lattices; siehe Abschnitt 3.3.3). Zum Beispiel dauerte die Entwicklung

des Beweises eines recht einfachen Schiebefensterprotokolls [Her92] unter Verwendung des Theorembeweisers OTTER [McC94] etwa einen Monat, wobei sich auch hier die Suche nach einer ausreichenden und beweisbaren Invariante als entscheidendes Problem herausstellte. Durch Berücksichtigung der Kompositionalität von aus Prozessen zusammengesetzten Systemen kann man die Komplexität von Verifikationen erheblich vereinfachen. Anstatt einen Einzelbeweis über das Gesamtsystem zu führen, weist man in mehreren Schritten die Korrektheit einzelner Subsysteme nach. Die Komplexität der zu betrachtenden Spezifikationen ist dann geringer, wodurch das Finden von Invarianten, Refinement Mappings und Lattices einfacher wird. Außerdem kann man bei einer derartigen Beweiszerlegung häufig Verifikationen aus zuvor bearbeiteten ähnlichen Projekten wiederverwenden.

Wie in Abschnitt 3.3.3 gezeigt wurde, entspricht der Beweis, daß ein durch eine Spezifikation Φ modelliertes System die Implementierung eines durch Ψ modellierten abstrakteren Systems darstellt, in TLA der Verifikation $\Phi \Rightarrow \Psi$. Wenn Φ und Ψ kompositionale Beschreibungen

$$\Phi_c \triangleq \Phi_1 \wedge \ldots \wedge \Phi_m \wedge \Phi CC \qquad \Psi_c \triangleq \Psi_1 \wedge \ldots \wedge \Psi_n \wedge \Psi CC$$

sind (siehe Abschnitt 4.1.2), kann der Verfeinerungsbeweis $\Phi_c \Rightarrow \Psi_c$ in Theoreme

$$\widehat{\Phi_l} \Rightarrow \Psi_l \tag{4.21}$$

zerlegen (siehe Abbildung 4.2), wobei $\widehat{\Phi_l}$ ein Subsystem von Φ_c ist. Außerdem muß die Konsistenz der Prozeßkopplungsbeschreibungen nachgewiesen werden:

$$\Phi CC \Rightarrow \Psi CC \tag{4.22}$$

Wie man ein Gesamtsystem Φ_c in Subsysteme $\widehat{\Phi_l}$ überführt, wird in Abschnitt 4.2.1 diskutiert. Bei Verfeinerungsbeweisen von Livenesseigenschaften muß immer das Gesamtsystem berücksichtigt werden, da Fairnessannahmen nur über „bedingte“ Aktionen getroffen werden, die von den Prozessen der Umgebung abhängen (siehe Abschnitte 4.1.1, 4.1.2). Livenessbeweise werden deshalb durch die Strukturierung nicht zwingend einfacher. Wir zeigen in Abschnitt 4.3.4 jedoch, daß bei einer geschickten Anordnung der fairen Aktionen in den Prozessen auch diese Beweise vereinfacht werden können. Der Verfeinerungsbeweis der Kopplungsbeschreibungen (Formel 4.22) ist im Prinzip komplexer, da der gesamte Zustandsraum von Φ_c und Ψ_c berücksichtigt werden muß. Jedoch profitiert er stark von den Beweisen der einzelnen Prozesse und kann häufig mechanisch ausgeführt werden (siehe Abschnitt 4.2.2). Die strukturierte Verifikation toleriert auch schrittweise Verfeinerungsbeweise. Existiert eine kompositionale Spezifikation Υ_c, kann der Beweis von $\Phi_c \Rightarrow \Psi_c$ in zwei Theoreme $\Phi_c \Rightarrow \Upsilon_c$ und $\Upsilon_c \Rightarrow \Psi_c$ zerlegt werden, die jeweils durch Verifikation der in 4.21 und 4.22 dargestellten Theoreme nachgewiesen werden.

4.2.1 Subsysteme

In Abschnitt 3.3.3 skizzierten wir die Struktur von Verfeinerungsbeweisen $\Phi \Rightarrow \Psi$. Zum einen muß die Initialbedingung von Φ auf diejenige von Ψ abgebildet werden (Formel 3.49). Zum anderen wird gezeigt, daß jede Aktion $\Phi.\pi_i$ des Feinsystems einer Aktion $\Psi.\pi_l$ oder einem Stotterschritt des Grobsystems entspricht (Formel 3.50). Weiterhin wird nachgewiesen, daß

Φ die Fairnessannahmen von Ψ erfüllt. Die Zustandsraum von Φ wird durch eine Zustandsfunktion auf denjenigen von Ψ abgebildet.

Wir erläuterten, daß man bei Anwendung des kompositionalen Stils Verfeinerungsbeweise in einfachere Theoreme zerlegen kann (siehe Formeln 4.21 und 4.22). Die Bildung eines geeigneten Subsystems $\widehat{\Phi_l}$ des Feinsystems Φ_c ist trivial, wenn ein Einzelprozeß Φ_{l_k} als Subsystem ausreicht:

$$\Phi_{l_k} \Rightarrow \Psi_l$$

Komplexere Subsysteme setzen sich aus mehreren Prozessen von Φ_c und einer Subsystemkopplungsbeschreibung $\Phi_l\widehat{CC}$ zusammen:

$$\widehat{\Phi_l} \triangleq \Phi_{l_1} \wedge \ldots \wedge \Phi_{l_m} \wedge \Phi_l\widehat{CC}$$

$\Phi_l\widehat{CC}$ ist eine Konjunktion aller Prozeßkopplungsbeschreibungen $P_{l_k}\widehat{CON}$ der Prozesse des Subsystems. Diese unterscheiden sich von den Prozeßkopplungsbeschreibungen $P_{l_k}CON$ aus ΦCC dadurch, daß in den Prädikaten $P_{l_k}.\widehat{\pi_i}PA$ und $P_{l_k}.\widehat{\pi_i}SR$ nur Konjunkte verwendet werden, die Aktionen von Prozessen Φ_{l_k} des Subsystems enthalten. Man sieht sofort, daß die Kopplungsbeschreibung des Subsystems aus der des Gesamtsystems folgt:

$$\Phi CC \Rightarrow \Phi_l\widehat{CC} \tag{4.23}$$

Bei der Definition von $\Phi_l\widehat{CC}$ wurde die ERV-Definitionsbeschreibung beibehalten, so daß in der monolithischen Beschreibung die Fairnessannahmen auch von Prozessen abhängen, die nicht zum Subsystem gehören. Das stellt jedoch kein Problem dar, da wir Subsysteme in dieser Form nur beim Beweis von Safetyeigenschaften einsetzen.

Zur Durchführung der Verfeinerungsverifikation wird ein Subsystem $\widehat{\Phi_l}$ nach der in Abschnitt 4.1.4 gezeigten Methode in ihre monolithische Form überführt.

4.2.2 Nachweis von Kopplungsbeschreibungen

Nachdem bewiesen wurde, daß alle Prozesse Ψ_l des Grobsystems Ψ durch Subsysteme $\widehat{\Phi_l}$ des Feinsystems Φ implementiert werden, bleibt noch der Nachweis, daß die Kopplungsbeschreibung ΦCC des Feinsystems diejenige des Grobsystems ΨCC erfüllt (siehe Formel 4.22). Das ist jedoch unter Berücksichtigung der Lemmata, die für die Verfeinerungsbeweise der Prozesse Ψ_l verwendet wurden, trivial. Dort wurde bewiesen, daß jede Aktion $\Psi_l.\pi_j$ durch ein Disjunkt in mindestens einer Aktionskopplungsbeschreibung $\Phi_{l_k}\widehat{CON}$ der Kopplungsbeschreibung $\Phi_l\widehat{CC}$ eines Subsystems impliziert wird. Aus der Definition von $\Phi_l\widehat{CC}$ folgt, daß es auch in einer Aktionskopplungsbeschreibung $\Phi_k\widehat{CON}$ ein Disjunkt gibt, für das

$$\Phi_k.\pi_i(p) \wedge \Phi_k.\pi_i PA(p) \wedge \Phi_k.\pi_i SR \Rightarrow \overline{\Psi_l.\pi_j} \tag{4.24}$$

gilt[7]. Dann bleibt aber nur noch nachzuweisen, daß alle Aktionen in den Prozessen Ψ_l, die miteinander gekoppelt sind, von demselben Disjunkt aus $\Phi_k\widehat{CON}$ impliziert werden. Wenn das der Fall ist, gilt

$$\forall l \exists k : \Phi_k CON \Rightarrow \Psi_l CON \tag{4.25}$$

[7]Der Oberstrich von $\overline{\Psi_l.\pi_j}$ zeigt an, daß der Zustandsraum von Ψ durch eine Zustandsfunktion auf den Zustandsraum von Φ abgebildet wurde (siehe Abschnitt 3.3.3).

Beim Verfeinerungsbeweis der Livenesseigenschaften wurde das gesamte Fein- und Grobsystem verwendet. Deshalb wird die ERV-Definitionsbeschreibung des Grobsystems durch die des Feinsystems erfüllt. Daraus folgt jedoch sofort, daß ΨCC durch ΦCC erfüllt wird.

Die Überprüfung beschränkt sich also auf eine Konsistenzprüfung der Aktionskopplungsbedingung, die mechanisiert werden kann.

4.3 Spezifikationen in cTLA$^+$

Die Spezifikationssprachen TLA$^+$ [Lam95] und tTLA$^+$ [Mes93], die in Abschnitt 3.4 vorgestellt wurden, können zur Beschreibung der Prozeßmodule und der Kopplungsbeschreibungen eingesetzt werden. Wir sind jedoch der Meinung, daß sie sich nicht gut für den Einsatz in realen Problemfeldern wie dem in Kapitel 5 vorgestellten Framework für Hochleistungstransportprotokolle eignen. Ein Grund dafür liegt in der TLA$^+$ zugrundeliegenden Sichtweise von Leslie Lamport, daß es im „Universum“ nur eine globale Menge an Variablen gibt, die allen Spezifikationen zugänglich ist. Deswegen bietet TLA$^+$ (und tTLA$^+$) keine Möglichkeit, Variablen in einem Modul zu kapseln und ihre aktuellen Werte vor der Modulumgebung zu verbergen. Das widerspricht jedoch dem Konzept des kompositionalen Spezifikationsstils, der private Variablen vorsieht, die nur innerhalb eines Einzelprozesses referenziert werden dürfen.

Ein zweiter Grund ist eher pragmatisch und liegt in der Umständlichkeit von TLA$^+$-Spezifikationen. TLA$^+$ und tTLA$^+$ sind sehr mächtige Spezifikationssprachen, mit denen man eine Vielzahl realer und theoretischer Systeme beschreiben kann. Aus diesem Grund fehlen ihnen syntaktische Konstrukte, die die Modellierbarkeit einschränken, die aber die Erstellung übersichtlicher und gut lesbarer Beschreibungen unterstützen[8]. So profitieren TLA$^+$-Spezifikationen nicht davon, daß sie fast immer in einer speziellen kanonischen Form erstellt werden (siehe Abschnitt 3.3.2).

Mit cTLA$^+$ [Hey95] wurde deshalb für den kompositionalen Spezifikationsstil eine neue Syntax eingeführt, die die Deklaration privater Variablen vorsieht und dem Sprachentwickler Konstrukte zur einfachen Spezifikation kanonischer Prozeßspezifikationen zur Verfügung stellt. Außerdem bietet cTLA$^+$ Konstrukte an, mit denen Operationen auf Prozeßspezifikationen wie die Komposition von Prozessen zu einem System einfach beschrieben werden können. Im Rahmen einer Diplomarbeit [Hey95] wurde ein Werkzeug entwickelt, das cTLA$^+$-Spezifikationen in tTLA$^+$-Beschreibungen überführt[9], so daß die in Abschnitt 3.5 vorgestellten Werkzeuge auch auf cTLA$^+$-Spezifikationen angewendet werden können. Außerdem kann das Werkzeug Operationen auf cTLA$^+$-Spezifikationen ausführen.

In Abschnitt 4.3.1 stellen wir die Syntax von cTLA$^+$ zur Beschreibung von Prozeßspezifikationen vor. Die cTLA$^+$-Operationen werden anschließend in Abschnitt 4.3.2 eingeführt. Lösungen zur Einhaltung der Stilkonventionen in cTLA$^+$-Spezifikationen beschreiben wir in Abschnitt 4.3.3. In Abschnitt 4.3.4 wird schließlich eine Spezifikationsmethode vorgestellt, die auch für Fairnesseigenschaften strukturierte Beweise ermöglicht.

4.3.1 cTLA$^+$-Prozeßtypspezifikationen

cTLA$^+$ ist eine Erweiterung von tTLA$^+$, dessen Syntax für konstante Module übernommen wird. Zur Beschreibung von Prozeßtypen, aus denen Prozeßinstanzen gebildet werden, wurde

[8] Ein Vergleich mit Assemblersprachen ist naheliegend.

[9] Dabei geht allerdings die Kapselung der privaten Variablen verloren.

ein neuer Modultyp eingeführt, der durch das Schlüsselwort `PROCESS` gekennzeichnet ist. Ein Beispiel hierfür ist die Spezifikation `P1`, die (leicht verändert) dem Framework für Hochleistungstransportprotokolle (siehe Kapitel 5) entnommen wurde:

```
PROCESS P1
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      cRq : key;
    INIT ≜ cRq = 0 ;
    ACTIONS
      Rq (krq : key) ≜ krq = cRq ∧ cRq′ = cRq + 1 ;
      In (krq : key) ≜ ( krq = "notsent" ∨ krq < cRq ) ∧ cRq′ = cRq ;
    WF: Rq;
END P1
```

Der Modultyp sieht die Einbindung von konstanten tTLA$^+$-Modulen vor (Schlüsselwörter `IMPORT` und `INCLUDE`). Boolesche Ausdrücke und Konstanten können wie bei tTLA$^+$-Modulen auch direkt vereinbart werden (Schlüsselwörter `BOOL` und `CONSTANT`). Die eigentliche Spezifikation eines Prozeßtyps ist in Feld `BODY` enthalten. `BODY` erlaubt einen kanonischen Aufbau von Prozeßbeschreibungen (siehe Abschnitt 4.1.1) durch fünf Unterfelder:

`VARIABLES:` Deklarationsteil für prozeßprivate Variablen. (Die Syntax lehnt sich an die Programmiersprachen PASCAL [JW78] und Modula [Wir85] an.)

`INIT` Initialprädikat des Prozesses.

`ACTIONS:` Folge von Aktionen (z.B. `Rq` und `In`), deren Disjunktion zusammen mit Stotterschritten die Zustandsübergangsrelation des Prozesses bildet.

`SF:` Liste von Bezeichnern der Aktionen aus dem Feld `ACTIONS`, deren „bedingte“ Aktionen stark fair sind.

`WF:` Liste von Bezeichnern der Aktionen aus dem Feld `ACTIONS`, deren „bedingte“ Aktionen schwach fair sind.

Das Feld `BODY` impliziert die Formel eines Prozeßtyps, die im Gegensatz zu TLA$^+$ und tTLA$^+$ nicht explizit aufgeführt werden muß. Für Prozeß `P1` lautet die in `BODY` spezifizierte Prozeßformel

$$P1 \triangleq \begin{cases} (cRq = 0) \wedge \\ \Box[\ \exists krq \in key : Rq(krq) \vee \\ \quad \exists krq \in key : In(krq)]_{(cRq)} \wedge \\ \forall krq \in key : WF_{(cRq)}(Rq(krq) \wedge (krq \in e_{Rq})) \end{cases}$$

Durch Verwendung generischer Prozeßtypparameter kann man mit einer Spezifikation eine Klasse an Prozeßtypen beschreiben. Die generischen Parameter werden in einem Klammerausdruck hinter dem Prozeßtypbezeichner deklariert[10]:

[10] Diese Schreibweise erscheint uns übersichtlicher als das in TLA$^+$ verwendete Feld `PARAMETERS`.

```
PROCESS P2 (usd : ANY;
            tc : SUBSET (usd × usd))
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      buf : SUBSET [[key, usd]] ;
    INIT ≜ buf = ∅ ;
    ACTIONS
      Rq (krq : key; d : usd) ≜
        buf′ = buf ∪ {[[krq,d]]} ;
      In (krq : key; d : usd) ≜
        ( krq = "notsent" ∨
          ∀ e : [[krq,e]] ∉ buf ∨
          ∃ e : [[krq,e]] ∈ buf ∧ tc(e,d) ) ∧
        buf′ = buf ;
    SF: Rq ;
    WF: In ;
END P2
```

In Prozeßbeispiel `P2` werden zwei Prozeßtypparameter `usd` und `tc` deklariert. Die durch diesen Prozeßtyp spezifizierte Prozeßformel hat die Form

$$P2(usd : ANY, tc : SUBSET(usd \times usd)) == \ldots$$

Ein Prozeßtyp kann durch Ersetzung der Parameter mit speziellen Werten instantiiert werden. Die Ersetzung von Parametern wird durch die cTLA$^+$-Operation der Prozeßverfeinerung vorgenommen (siehe Abschnitt 4.3.2).

4.3.2 cTLA$^+$-Prozeßoperationen

Im Verlauf der Entwicklung eines realen Systems müssen cTLA$^+$-Prozeßtypspezifikationen auf verschiedenartige Weise umgeformt werden. Beispiele für Umformungen sind die in der Abbildung 4.3 skizzierten Kompositionen, Verfeinerungen und Dekompositionen von Spezifikationen. Um diese Umformungen zu vereinfachen, enthält die Syntax von cTLA$^+$ Prozeßoperationen, mit denen man Prozeßtypen umformen kann. Durch das in [Hey95] vorgestellte Werkzeug kann man diese Umformungen ausführen lassen. Die folgenden Operationen werden durch cTLA$^+$ unterstützt:

- Prozeßkomposition,
- Prozeßverfeinerung,
- Prozeßintegration,
- Prozeßkombination,
- Prozeßzerlegung,

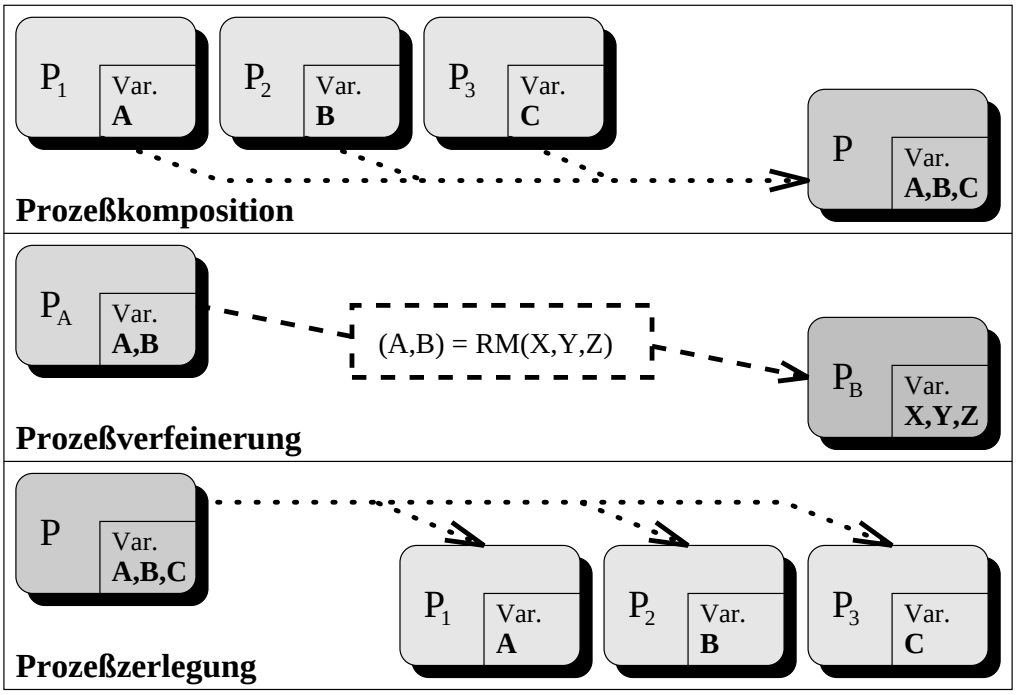

Abbildung 4.3: Umformungen auf cTLA$^+$-Prozessen

- Aktionszerlegung und
- Prozeßfelder.

Die Prozeßkomposition entspricht der Komposition mehrerer Prozeßtypen zu einer Systembeschreibung. Die Prozeßverfeinerung erlaubt die Ersetzung der Variablen eines Prozeßtyps durch Zustandsfunktionen, die Variablen aus einem anderen Zustandsraum enthalten. Die Prozeßintegration ist eine Kombination aus der Prozeßkomposition und der Prozeßverfeinerung. Eine vereinfachte Darstellung spezieller Prozeßkompositionen und -integrationen, bei der auf die Angabe spezieller Prozeßkopplungsbeschreibungen und Variablenersetzungsbeschreibungen verzichtet werden kann, wird durch die Prozeßkombination ermöglicht. Die Prozeßzerlegung ist eine Umkehrung der Prozeßkomposition und erlaubt die Zerlegung eines Prozeßtyps in mehrere Unterprozeßtypen, deren Komposition den ursprünglichen Prozeßtyp ergeben. Aktionen eines Prozeßtyps kann man mit Hilfe der Aktionszerlegung in mehrere Aktionen zerlegen. Die cTLA$^+$-Operation der Prozeßfelder erlaubt schließlich, einen Prozeßtyp zu definieren, der aus der Komposition einer beliebig großen Menge gleichartige Prozeßtypen besteht.

Als Beispiel für eine Prozeßkomposition betrachten wir die Spezifikation `ComposedP`:

```
PROCESS ComposedP (usd : ANY;
                   tc1 : SUBSET (usd × usd);
                   tc2 : SUBSET (usd × usd) )
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
```

```
  BODY
    PROCESSES
      CP_1 : P1;
      CP_2 : P2(usd,tc1);
      CP_3 : P2(usd,tc2);
    ACTIONS
      Rq (krq : key; d : usd) ≙
        CP_1.Rq(krq) ∧ CP_2.Rq(krq,d) ∧ CP_3.Rq(krq,d) ;
      In1 (krq : key; d : usd) ≙
        CP_1.In(krq) ∧ CP_2.In(krq,d) ∧ CP_3.Stutter ;
      In2 (krq : key; d : usd) ≙
        CP_1.Stutter ∧ CP_2.Stutter ∧ CP_3.In(krq,d) ;
END ComposedP
```

Die Syntax dieser Operation ist mit Ausnahme des Feldes `BODY` identisch zu einfachen Prozeßtypspezifikationen. `BODY` besitzt zwei Unterfelder zur Beschreibung der miteinander zu komponierenden Prozesse und der Aktionskopplung:

`PROCESSES`: Deklaration der zu komponierenden Prozeßtypen. Im aktuellen Zustand des Werkzeuges [Hey95] müssen die Prozeßtypbezeichner (z.B. `P2`) monolithische Prozeßtypspezifikationen sein. Da Spezifikationen mehrfach komponiert werden können (z.B. `P2`), wird jedem Prozeßtyp ein eindeutiger Bezeichner zugeordnet (z.B. $\texttt{CP}_2$). Die Prozeßtypparameter sind Ausdrücke, die aus vereinbarten Konstanten und den Prozeßtypparametern des komponierten Systems (z.B. `tc1`) gebildet werden.

`ACTIONS`: Festlegung der Aktionskopplungen. Das Feld besteht aus Vereinbarungen von Systemaktionen, die als Konjunktion von Prozeßaktionen definiert sind. Den Prozeßaktionen wird der eindeutige Prozeßtypbezeichner (z.B. $\texttt{CP}_2\texttt{.In(krq,d)}$) als Präfix zugeordnet. Prozesse, die bei der Schaltung der Systemaktion Stotterschritte ausführen, werden durch den Ausdruck `Stutter` markiert.

Diese Schreibweise erspart die mühselige Erstellung der Kopplungsbeschreibung CC. Das Werkzeug garantiert die Stilkonventionen $C_1, \ldots, C_5$ sowie E_1 und E_2[11]. Zur Verdeutlichung seiner Arbeitsweise geben wir im folgenden die vom Werkzeug erzeugte monolithische Spezifikation von `ComposedP` an:

```
PROCESS ComposedP (usd : ANY;
                   tc1 : SUBSET (usd × usd);
                   tc2 : SUBSET (usd × usd) )
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS key ≙ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      CP_1cRq : key;
      CP_2buf : set of [[key, usd]] ;
      CP_3buf : set of [[key, usd]] ;
    INIT ≙ CP_1cRq = 0 ∧ CP_2buf = ∅ ∧ CP_3buf = ∅ ;
    ACTIONS
```

[11] Die Einhaltung der Stilkonventionen P_1 und P_2 obliegt dem Spezifizierer.

```
    Rq (krq : key; d : usd) ≜
      krq = CP1cRq ∧
      CP1cRq′ = CP1cRq + 1 ∧
      CP2buf′ = CP2buf ∪ {[[krq,d]]} ∧
      CP3buf′ = CP3buf ∪ {[[krq,d]]} ;
    In1 (krq : key; d : usd) ≜
      ( krq = "notsent" ∨ krq < CP1cRq ) ∧
      CP1cRq′ = CP1cRq ∧
      ( krq = "notsent" ∨
        ∀ e : [[krq,e]] ∉ CP2buf ∨
        ∃ e : [[krq,e]] ∈ CP2buf ∧ tc1(e,d) ) ∧
      CP2buf′ = CP2buf ∧
      CP3buf′ = CP3buf ;
    In2 (krq : key; d : usd) ≜
      CP1cRq′ = CP1cRq ∧
      CP2buf′ = CP2buf ∧
      ( krq = "notsent" ∨
        ∀ e : [[krq,e]] ∉ CP3buf ∨
        ∃ e : [[krq,e]] ∈ CP3buf ∧ tc2(e,d) ) ∧
      CP3buf′ = CP3buf ;
  SF : Rq ;
  WF : In1, In2 ;
END ComposedP
```

Als Fairnessannahmen des komponierten Prozeßtyps werden die jeweils stärksten Fairnessanforderungen der Einzelprozeßtypen verwendet.

Die cTLA$^+$-Operation Prozeßverfeinerung erleichtert Verfeinerungsbeweise, da sie die Anpassung des Zustandsraums der Grobspezifikation Ψ an denjenigen der Feinspezifikation Φ unterstützt (siehe Formel 3.45 in Abschnitt 3.3.3). Ψ wird in eine Spezifikation $\widehat{\Psi}$ überführt, die den gleichen Variablensatz wie Φ besitzt. Bei kleineren Verfeinerungsschritten kann die Überprüfung, daß Φ eine Implementierung von $\widehat{\Psi}$ ist, sogar automatisch erfolgen, indem $\widehat{\Psi}$ durch einfache Termersetzungsschritte (siehe auch Abschnitt 3.1.5) nach Φ übersetzt wird. Im folgenden wird eine Verfeinerungsoperation von Prozeßtyp `P1` gezeigt:

```
PROCESS RefineP1
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  BODY
    VARIABLES
      c1 : key;
      c2 : Nat;
    REFINE P1
      SUBSTITUTE
        cRq BY IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2 ;
END RefineP1
```

Das Feld `BODY` bei cTLA$^+$-Verfeinerungsoperationen hat zwei Unterfelder:

`VARIABLES:` Deklaration der Variablen, die nach der Ersetzung verwendet werden sollen.

`REFINE:` Angabe des zu verfeinernden Prozesses. In einem Unterfeld `SUBSTITUTE` wird angegeben, durch welche Zustandsfunktionen die Variablen des zu verfeinernden Prozesses ersetzt werden.

In unserem Beispiel soll die Variable `cRq` aus `P1` durch die Variablen `c1` und `c2` ersetzt werden, die im Feld `VARIABLES` vereinbart sind. Wenn `cRq` den Wert `'notsent'` hat, soll `c1` ebenfalls `'notsent'` und `c2` den Wert 22 annehmen. In allen anderen Fällen soll `cRq` der Summe von `c1` und `c2` entsprechen. Diese Ersetzung führt zum folgenden Prozeßtyp:

```
PROCESS RefineP1
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      c1 : key;
      c2 : Nat ;
    INIT ≜ (IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2) = 0 ;
    ACTIONS
      Rq (krq : key) ≜
        krq = (IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2) ∧
        (IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2)′ =
        (IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2) + 1 ;
      In (krq : key) ≜
        ( krq = "notsent" ∨
          krq < (IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2) ∧
          (IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2)′ =
          (IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2) ;
    WF: Rq;
END RefineP1
```

Das Ergebnis dieser Ersetzung ist nicht sehr übersichtlich. Häufig kann man jedoch durch einfache Ersetzungen eine leichter lesbare Form finden. In unserem Beispiel kann man nachweisen, daß das `IF`-Konstrukt niemals den Wert `'notsent'` annimmt. Deshalb kann es durch den einfacheren Term `(c1 + c2)` ersetzt werden.

Durch Prozeßverfeinerungen entstehende Prozeßtypen können unerfüllbar sein, da man die Variablen so ersetzen kann, daß die Initialbedingungen des erzeugten Prozeßtyps falsch sind.

Prozeßintegration ist eine Kombination der cTLA$^+$-Operationen der Prozeßkomposition und der Prozeßverfeinerung. Sie wurde als eigenständige Prozeßtypoperation eingeführt, weil sie im Ansatz, den Spezifikationsstil zur korrektheitserhaltenden Konstruktion verteilter Anwendungen einzusetzen (siehe Kapitel 7 und vgl. [MK95]), von großer Bedeutung ist. In einem Beispiel wird die Integration von Prozeßtyp `P1` mit zwei Duplikaten des Prozeßtyps `P2` gezeigt:

```
PROCESS IntegrateP (usd : datatype;
                    tc1 : SUBSET (usd × usd);
                    tc2 : SUBSET (usd × usd) )
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
```

```
    VARIABLES
      c1 : key;
      c2 : Nat ;
      abuf : [{ 0, 1} → SUBSET([[key, usd]]) ] ;
    INTEGRATE
      IP_1 : P1
        SUBSTITUTE IP_1.cRq BY
          IF (c1 = "notsent" ∧ c2 = 22) THEN "notsent" ELSE c1 + c2;
      IP_2 : P2(usd,tc1)
        SUBSTITUTE IP_2.buf BY abuf[0] ;
      IP_3 : P2(usd,tc2)
        SUBSTITUTE IP_3.buf BY abuf[1] ;
    ACTIONS
      Rq (krq : key; d : usd) ≜
        IP_1.Rq(krq) ∧ IP_2.Rq(krq,d) ∧ IP_3.Rq(krq,d) ;
      In1 (krq : key; d : usd) ≜
        IP_1.In(krq) ∧ IP_2.In(krq,d) ∧ IP_3.Stutter ;
      In2 (krq : key; d : usd) ≜
        IP_1.Stutter ∧ IP_2.Stutter ∧ IP_3.In(krq,d) ;
END IntegrateP
```

Das Feld `BODY` dieser Operation besteht aus drei Unterfeldern:

`VARIABLES`: Deklaration der Variablen, die nach der Ersetzung verwendet werden sollen.

`INTEGRATE`: Angabe der zu komponierenden Prozeßtypen mit der Zuweisung eines eindeutigen Bezeichners (z.B. $\texttt{IP}_2$). In einem Unterfeld `SUBSTITUTE` wird für jeden Prozeßtyp angegeben, durch welche Zustandsfunktionen die Variablen des Prozeßtyps ersetzt werden.

`ACTIONS`: Festlegung der Aktionskopplungen. Das Feld besteht aus Vereinbarungen von Systemaktionen, die als Konjunktion von Prozeßaktionen definiert sind. Den Prozeßaktionen wird der eindeutige Prozeßtypbezeichner (z.B. $\texttt{IP}_2\texttt{.In(krq,d)}$) als Präfix zugeordnet. Prozesse, die bei der Schaltung einer Systemaktion Stotterschritte ausführen, werden durch den Ausdruck `Stutter` markiert.

Die cTLA$^+$-Operation Prozeßkombination wurde eingeführt, um die Komposition bzw. Integration der Prozeßtypen im Framework für Hochleistungsprotokolle (siehe Kapitel 5) zu vereinfachen. Sie erlaubt wesentlich einfachere Kopplungsbeschreibungen als die Prozeßkomposition, kann jedoch nur eingesetzt werden, wenn man sich bei der Beschreibung der zu komponierenden Prozeßtypen an die folgenden Namenskonventionen hält:

- Alle Aktionen in den Prozeßtypen, die miteinander gekoppelt werden sollen, besitzen denselben Aktionsbezeichner.
- Die Parameter von Aktionen mit gleichem Namen, die bei Kopplungen einander zugeordnet werden, besitzen ebenfalls dieselben Bezeichner.
- Variablen mit gleichem Namen in verschiedenen Prozeßtypen werden im kombinierten Prozeßtyp durch eine Variable ersetzt. Deshalb müssen alle Variablen mit dem gleichen Namen dasselbe Verhalten modellieren.

Das folgende Beispiel veranschaulicht diese Operation:

```
PROCESS CombineP (usd : datatype ;
                  tc1 : SUBSET (usd × usd) ;
                  tc2 : SUBSET (usd × usd) )
  COMBINE
    CoP1 : P1;
    CoP2 : P2(usd,tc1);
    CoP3 : P2(usd,tc2);
END CombineP
```

Die Prozeßkombination verwendet nur ein Feld `COMBINE`, in dem die miteinander zu kombinierenden Prozeßtypen mit ihren Parametern und eindeutigen Bezeichern (z.B. $\texttt{CoP}_1$) definiert sind.

Die Operation sieht vor, daß alle Aktionen mit demselben Bezeichner miteinander gekoppelt sind. Ferner sichert sie vollständiges Interleaving zu. Alle Variablen mit demselben Bezeichner werden durch eine einzige Variable ersetzt. Zur Verdeutlichung geben wir die Prozeßintegrationsoperation an, der die oben angeführte Kombination entspricht:

```
PROCESS CombineP (usd : datatype;
                  tc1 : SUBSET (usd × usd);
                  tc2 : SUBSET (usd × usd) )
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≙ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      cRq : nat ;
      buf : set of [[key, usd]] ;
    INTEGRATE
      CoP1 : P1
        SUBSTITUTE CoP1.cRq BY cRq ;
      CoP2 : P2(usd,tc1)
        SUBSTITUTE CoP2.buf BY buf ;
      CoP3 : P2(usd,tc2)
        SUBSTITUTE CoP3.buf BY buf ;
    ACTIONS
      Rq (krq : key; d : usd) ≙
        CoP1.Rq(krq) ∧ CoP2.Rq(krq,d) ∧ CoP3.Rq(krq,d) ;
      In (krq : key; d : usd) ≙
        CoP1.In(krq) ∧ CoP2.In(krq,d) ∧ CoP3.In(krq,d) ;
END CombineP
```

Eine weitere wichtige Prozeßoperation ist die Zerlegung einer monolithischen Prozeßtypspezifikation in eine Komposition aus Unterprozeßtypen. Der Spezifizierer kann neue Prozesse erzeugen, die jeweils nur einen Teil der privaten Variablen des Ausgangsprozesses besitzen, wobei auch die Zuordnung einer Variablen zu mehreren Unterprozessen erlaubt ist. In der Initialbedingung und den Aktionen eines Subprozesses werden nur Terme des ursprünglichen Prozesses belassen, die ausschließlich in diesem Subprozeß definierte Variablen enthalten. Gemischte Terme mit Variablen aus mehreren Subprozessen werden von dem in [Hey95]

vorgestellten Werkzeug, um eine größere Flexibilität zu erhalten, in die Initialbedingung bzw. in die Aktionen eines zusätzlichen Prozesses geschrieben. Ein weiterer Prozeß wird für konstante Terme eingeführt. Zur Illustration führen wir einen neuen Prozeßtyp `P3` ein:

```
PROCESS P3 (usd : datatype)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      a, b : key;
    INIT ≜ a = 0 ∧ b = 1 ;
    ACTIONS
      Rq (krq : key) ≜ a < b ∧ a = krq ∧ krq > 0 ∧ a′ = a + 1 ∧ b′ = b ;
      In (krq : key) ≜ ( a ≥ b ∨ a ≠ krq ) ∧ b′ = b + 1 ∧ a′ = a ;
    WF: Rq;
END P3
```

Die Zerlegung dieses Prozeßtyps erfolgt durch die Prozeßzerlegungsoperation `SplitP3`, die zwei Unterprozeßtypen `SP1` und `SP2` erzeugt:

```
PROCESS SplitP3 (usd : datatype)
  SPLIT PROCESS P3
    PROCESSES
      SP1 : SP1(usd) HAS a;
      SP2 : SP2(usd) HAS b;
END SplitP3
```

Diese Operation hat ein Feld `SPLIT PROCESS`, in dem der zu zerlegende Prozeß angegeben wird. In einem Unterfeld `PROCESSES` werden die zu erzeugenden Unterprozeßtypen einschließlich der zu verwendenden Parameter und der Zerlegung der Variablen definiert. In unserem Beispiel erhält der Prozeßtyp `SP1` die Variable `a` und `SP2` die Variable `b`. Die eindeutigen Bezeichner (z.B. $\mathtt{SP}_1$) dienen nur zur Zuordnung in der Integrationsbeschreibung (siehe unten).

`SP1` und `SP2` besitzen die folgende Form:

```
PROCESS SP1 (usd : datatype)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      a : key;
    INIT ≜ a = 0 ;
    ACTIONS
      Rq (krq : key) ≜ a = krq ∧ a′ = a + 1 ;
      In ≜ a′ = a ;
    WF: Rq;
END SP1
```

```
PROCESS SP2 (usd : datatype)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      b : key;
    INIT ≜ b = 1 ;
    ACTIONS
      Rq ≜ b′ = b ;
      In ≜ b′ = b + 1 ;
    WF: Rq;
END SP2
```

Der gemischte Term `a < b` in Aktion `P3.Rq` wird in keiner dieser Prozeßtypen verwendet, sondern befindet sich in der Aktion `Rq` eines zusätzlichen Prozeßtyps `P3Mixed`, der beide Variablen `a` und `b` enthält. Der konstante Ausdruck `krq > 0` aus Aktion `P3.Rq` wird in einem weiteren Prozeßtyp `P3Const` eingesetzt, in dem keine Variablen vereinbart sind. Sind keine gemischten bzw. konstanten Ausdrücke vorhanden, werden diese zusätzlichen Prozesse nicht erzeugt.

Außerdem erzeugt das Werkzeug die Integrationsoperation, mit der man aus den Unterprozeßtypen `SP1`, `SP2`, `P3Mixed` und `P3Const` wieder den originalen Prozeßtyp `P3` bildet:

```
PROCESS SplitP3 (usd : datatype)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      a, b : key;
    INTEGRATE
      SP_1 : SP1(usd)
        SUBSTITUTE SP_1.a BY a ;
      SP_2 : SP2(usd)
        SUBSTITUTE SP_2.b BY b ;
      M : P3Mixed(usd)
        SUBSTITUTE M.a BY a, M.b BY b ;
      C : P3Const(usd) ;
    ACTIONS
      Rq (krq : key) ≜
        SP_1.Rq(krq) ∧ SP_2.Rq ∧ M.Rq ∧ C.Rq(krq) ;
      In (krq : key) ≜
        SP_1.In ∧ SP_2.In ∧ M.In(krq) ;
END SplitP3
```

Nach der Stilkonvention C_3 darf eine faire Aktion eines Einzelprozesses nur in einer einzigen Aktionskopplung enthalten sein. Durch Aufspaltung von Aktionen in mehrere sich gleich verhaltende Aktionen kann diese Einschränkung jedoch umgangen werden[12].

[12] Allerdings wird dadurch die Stilkonvention P_1 verletzt (Lösung siehe Abschnitt 4.3.3).

Die Zerlegung von Aktionen wird durch die cTLA$^+$-Operation der Aktionszerlegung unterstützt. Eine Aktion kann in mehrere Aktionen mit unter Umständen unterschiedlichen Schaltbedingungen zerlegt werden:

```
PROCESS SplitActP3 (usd : datatype)
  SPLIT ACTIONS P3
    ACTIONS
      Rq TO
        Rq1 IF odd(b);
        Rq2 IF even(b);
      In TO
        In1 IF a ≥ 7000;
        In2 IF a ≤ 5000;
        In3 IF True;
END SplitActP3
```

Der Prozeßtyp besitzt ein Feld `SPLIT ACTIONS`, mit dem der Ausgangsprozeß bezeichnet wird. In einem Unterfeld `ACTIONS` werden die zu zerlegenden Aktionen, die Bezeichner der neuen Aktionen und die erweiterten Schaltbedingungen dieser Aktionen angegeben. Die neuen Aktionen besitzen die gleichen Fairnessannahmen wie ihr Original.

Das Ergebnis der Aktionszerlegung ist nicht zwingend eine Verfeinerung, da man die Schaltbedingungen fairer Aktionen so weit einschränken kann, daß aus der Fairnessannahme nicht mehr die Fairness der Aktion des Ausgangssystems geschlossen werden kann.

Der folgende Prozeßtyp wird durch die cTLA$^+$-Aktionszerlegung aus dem oben genannten Beispiel erzeugt:

```
PROCESS SplitActP3 (usd : datatype)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      a, b : key;
    INIT ≜ a = 0 ∧ b = 1 ;
    ACTIONS
      Rq1 (krq : key) ≜
        odd(b) ∧ a < b ∧ a = krq ∧ krq > 0 ∧ a′ = a + 1 ∧ b′ = b ;
      Rq2 (krq : key) ≜
        even(b) ∧ a < b ∧ a = krq ∧ krq > 0 ∧ a′ = a + 1 ∧ b′ = b ;
      In1 (krq : key) ≜
        a ≥ 7000 ∧ (a ≥ b ∨ a ≠ krq) ∧ b′ = b + 1 ∧ a′ = a ;
      In2 (krq : key) ≜
        a ≤ 5000 ∧ (a ≥ b ∨ a ≠ krq) ∧ b′ = b + 1 ∧ a′ = a ;
      In3 (krq : key) ≜
        ( a ≥ b ∨ a ≠ krq ) ∧ b′ = b + 1 ∧ a′ = a ;
    WF: Rq1, Rq2;
END SplitActP3
```

Die Komposition einer beliebig großen Anzahl an Duplikaten eines Prozeßtyps, dessen Prozesse zueinander nebenläufig sind, kann mit Hilfe des cTLA$^+$-Konstrukts Prozeßfelder erzeugt werden:

```
PROCESS ArrayP3 (n : nat)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    ARRAY
      i : {x | 0 ≤ x ≤ n }
      OF
        ar : P3({ x | 0 ≤ x ≤ i });
END ArrayP3
```

Das Feld `BODY` enthält ein Unterfeld `ARRAY`. Dort wird der Wertebereich des Prozeßfeldes definiert und ein Konstantenbezeichner (z.B. `i`) angegeben. Hinter dem Schlüsselwort `OF` wird der zu duplizierende Prozeßtyp vereinbart und mit einem eindeutigen Bezeichner versehen. Der Konstantenbezeichner des Wertebereichs darf in den Prozeßtypparameterdefinitionen verwendet werden, so daß die Elemente des Prozeßfeldes unterschiedliche Ausprägungen haben können. In unserem Fall entspricht des Prozeßfeld der Komposition der Prozeßtypen `P3(0)`,`P3(1)`,...,`P3(n)`. Dabei werden vom Werkzeug anstatt jeweils $n+1$ Variablen `a` und `b` zwei Feldvariablen `ara` und `arb` erzeugt. Das System hat $n+1$ Systemaktionen `Rq` und `In`. Diese Aktionen werden im Prozeßfeld durch zwei Aktionen `Rq` und `In` spezifiziert, die jeweils einen zusätzlichen Parameter `i` aus der Menge `{x | 0 ≤ x ≤ n }` besitzen.

Das Ergebnis der cTLA$^+$-Operation ist in unserem Beispiel der Prozeßtyp `ArrayP3`:

```
PROCESS ArrayP3 (usd : datatype)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      ara, arb : [ {x | 0 ≤ x ≤ n } → key ];
    INIT ≜ ∀ i : {x | 0 ≤ x ≤ n } ⇒ a[i] = 0 ∧ b[i] = 1 ;
    ACTIONS
      Rq (i : {x | 0 ≤ x ≤ n }; krq : key) ≜
        ara[i] < arb[i] ∧ ara[i] = krq ∧ krq > 0 ∧
        ara′ = [ara EXCEPT i ↦ ara[i] + 1] ∧ arb′ = arb ;
      In (i : {x | 0 ≤ x ≤ n }; krq : key) ≜
        ( ara[i] ≥ arb[i] ∨ ara[i] ≠ krq ) ∧
        arb′ = [arb EXCEPT i ↦ arb[i] + 1] ∧ ara′ = ara ;
    WF: Rq;
END ArrayP3
```

4.3.3 Einhaltung der Stilkonventionen

In Kapitel 4.1.3 wurden Stilkonventionen vorgestellt, deren Einhaltung für den Beweis notwendig sind, daß aus Prozessen komponierte Systeme erfüllbar sind und in eine monolithische Form überführt werden können. Leider schränken die Stilkonventionen P_1, P_2 und C_3 die Expressivität des kompositionalen Stils ein. Durch P_1 und P_2 wird ausgeschlossen, daß eine

faire Aktion dieselben Zustandsübergänge wie eine andere Aktion des gleichen Prozesses haben kann. Durch C_3 wird zudem verhindert, daß eine faire Prozeßaktion in mehr als einer Systemaktion gekoppelt wird. Um diese Einschränkungen umgehen zu können, stellen wir eine einfache Methode vor, durch die der Zustandsraum so vergrößert wird, daß alle fairen Aktionen unterschiedliche Zustandsübergänge bezeichen.

Die folgende Abbildung zeigt einen Prozeß H:

```
PROCESS H
  IMPORT Natural ;
  BODY
    VARIABLES
      v : { 0, 1 };
    INIT ≙ v = 0 ;
    ACTIONS
      ah ≙ v' = 1 - v;
END H
```

Eine binäre Variable v besitzt im Initialzustand den Wert 0 und wird bei jeder Schaltung der Prozeßaktion ah zwischen 0 und 1 umgeschaltet.

Ein Prozeßtyp P, dessen Aktionen die Stilkonventionen P_1 bzw. P_2 nicht erfüllen, wird durch Komposition mit H in einen Prozeßtyp PC überführt, dessen Aktionen zu P_1 und P_2 konform sind. Wenn P n-viele Aktionen $\mathtt{P}.\pi_i$ besitzt, wird er mit n-vielen Instanzen $\mathtt{H}_i$ von H komponiert. Dabei wird jede faire Aktion $\mathtt{P}.\pi_i$ mit der Aktion $\mathtt{H}_i.\mathtt{ah}$ gekoppelt. Die Variablen der anderen Instanzen stottern bei der Schaltung von $\mathtt{P}.\pi_i$.

Das Resultat der Komposition PC erfüllt die Stilkonvention P_1, da $\mathtt{H}_i.\mathtt{v}$ nur bei Ausführung von Aktion $\mathtt{PC}.\pi_i$ umschaltet. P_2 wird ebenfalls erfüllt, da die Variable $\mathtt{H}_i.\mathtt{v}$ bei jeder Schaltung von $\mathtt{PC}.\pi_i$ einen anderen Wert annimmt und deshalb niemals stottert. Somit ist diese Aktion zu allen anderen Aktionen disjunkt. Das nach außen sichtbare Verhalten von PC unterscheidet sich nicht von P, da die Variablen $\mathtt{H}_i.\mathtt{v}$ Vergangenheitsvariablen sind (siehe Abschnitt 3.3.4), von denen keine Schaltbedingungen abhängen. In den Implementierungen werden diese Variablen einfach weggelassen.

Will man eine faire Aktion $\mathtt{P}.\pi_i$ in mehreren Systemaktionen einsetzen, dupliziert man sie zunächst durch die cTLA$^+$-Aktionszerlegung. Anschließend komponiert man das Resultat mit Instanzen von H. Nun erfüllen die Duplikate die Stilkonvention P_2, obwohl die relevanten Variablen dieselben Zustandsübergänge beschreiben. Jedes Duplikat wird in genau einer Systemaktion verwendet.

4.3.4 Strukturierung von Fairnessbeweisen

Im Gegensatz zu den Safetybeweisen können Fairnessbeweise in zusammengesetzten Systemen nicht in Einzelbeweise zerlegt werden, die nur von Verhaltensfolgen einzelner Prozesse abhängen (siehe Abschnitt 4.2). Strukturierte Fairnessbeweise bringen also gegenüber herkömmlichen Beweisen keine Vorteile. Wir zeigen jedoch, daß man unter Einhaltung eines bestimmten Stils bei der Spezifikation der Prozeßtypen eines modellierten Systems auch die Verifikation von Fairnesseigenschaften vereinfachen kann. Aus Gründen der Orthogonalität empfehlen wir, Safetyeigenschaften und Fairnesseigenschaften immer in separaten Prozessen zu spezifizieren. Ein Beispiel einer Prozeßspezifikation, die nicht die Safetyeigenschaften ihrer Umgebung beeinflußt, ist der Prozeß L:

```
PROCESS L (usd : datatype)
  IMPORT Natural ;
  INCLUDE Constants WITH Ctyp ← usd ;
  CONSTANTS
    key ≜ nat ∪ {"notsent"} ;
  BODY
    VARIABLES
      a, b : key;
    INIT ≜ a = 0 ∧ b = 1 ;
    ACTIONS
      fRq (krq : key) ≜ a < b ∧ a = krq ∧ a′ = a + 1 ∧ b′ = b ;
      nRq (krq : key) ≜ (a ≥ b ∨ a ≠ krq) ∧ a′ = a + 1 ∧ b′ = b ;
      In ≜ b′ = b + 1 ∧ a′ = a ;
    WF: fRq;
END L
```

Prozesse dieser Art, sogenannte Fairnessprozesse, besitzen nur Aktionen ohne Fairnessannahme, die keine Schaltbedingungen haben (z.B. `In`), oder Paare von Aktionen, von denen eine Aktion fair ist (z.B. `fRq`, `nRq`). Die beiden Aktionen eines Paares beschreiben dieselben Zustandsübergänge, besitzen jedoch zueinander komplementäre Schaltbedingungen.

Eine der beiden Aktionen eines Paares ist immer schaltbar. Wenn man diese Disjunktion als eine Aktion betrachtet, besteht die Zustandsübergangsrelation des Fairnessprozesses nur aus Aktionen, die keine Schaltbedingungen besitzen. Da seine Variablen privat sind (siehe Abschnitt 3.3.4), beeinflußt er nicht die Safetyeigenschaften der mit ihm komponierten Prozesse, sondern nur deren Livenesseigenschaften.

Will man eine Aktion $\texttt{P}_i.\pi_j$ eines Safetyprozesses mit `L.fRq` koppeln, um sie mit der Fairnessanforderung von `L.fRq` zu versehen, erzeugt man eine faire Systemaktion aus $\texttt{P}_i.\pi_j$ und `L.fRq` sowie eine nicht faire Systemaktion aus $\texttt{P}_i.\pi_j$ und `L.nRq`[13]. Das aus der Komposition von $\texttt{P}_i$ und `L` bestehende System hat dieselben Safetyeigenschaften wie $\texttt{P}_i$.

Man legt bei der Entwicklung der Fairnessprozesse die Schaltbedingungen jeder fairen Aktion so stark aus, daß sie nur schaltbar ist, wenn auch die im System mit ihr gekoppelten Aktionen der anderen Prozesse geschaltet werden dürfen:

$$\Box(\forall p : \mathit{Enabled}(L.\pi_j(p)) \Rightarrow p \in L.e_{\pi_j}) \tag{4.26}$$

Im Fairnessprozeß ist also ein im allgemeinen stark vereinfachtes Modell seiner aus Safetyprozessen bestehenden Umgebung enthalten. Anstatt des Gesamtsystems muß bei einem Livenessbeweis nur der Fairnessprozeß und das in ihm enthaltene Modell der Safetyprozesse berücksichtigt werden, so daß die Beweiskomplexität meist erheblich reduziert ist.

Ein Beispiel ist die Komposition von Prozeß `P3` ohne dessen Fairnessanforderung (siehe Abschnitt 4.3.2) mit `L`. Dabei wird Aktion `P3.Rq` sowohl mit der Aktion `L.fRq` als auch mit `L.nRq` gekoppelt. Man sieht sofort, daß die Variablen `P3.a` und `P3.b` immer die gleichen Werte wie `L.a` bzw. `L.b` haben, so daß `P3.Rq(krq)` immer schaltbar ist, wenn auch `L.fRq(krq)` geschaltet werden kann. Daraus folgt die Formel 4.26 für `fRq`.

Die Vereinfachung der Livenessbeweise wird anhand eines abstrakten Szenarios in Abbildung 4.4 gezeigt. Ein Feinsystem F besteht aus den Safetyprozessen $F_1, \ldots, F_4$ und einem

[13] Da $\texttt{P}_i.\pi_j$ nicht fair ist, wird durch diese Kopplung nicht die Stilkonvention C_3 verletzt.

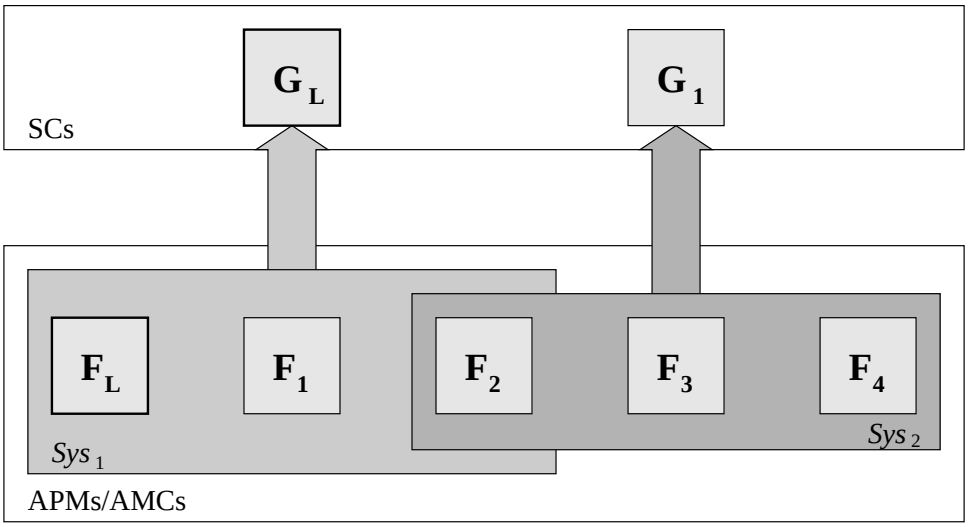

Abbildung 4.4: Szenario eines Fairnessbeweises (entnommen aus [HK95])

Fairnessprozeß F_L. Die fairen Aktionen $F_L.\pi_j$ haben so starke Schaltbedingungen, daß Formel 4.26 für beliebige Komposition von F_L mit den Safetyprozessen von F gilt. Daraus folgt dann

$$\Box(\forall p : Enabled(L.\pi_j(p)) \Rightarrow Enabled(L.e\pi_j(p))) \tag{4.27}$$

Bei der Durchführung der Livenessverifikation von Prozeß G_L müssen nur die Prozesse F_L, F_1 und F_2 des Subsystems SF_1 berücksichtigt werden. Wie bei den Safetybeweisen ist dadurch aber die Komplexität im Vergleich zu einer Verifikation des Gesamtsystems reduziert.

Kapitel 5

Ein Framework für Hochleistungstransportprotokolle

Eine Methode, die formale Entwicklung im Arbeitsgebiet der Hochleistungstransportprotokolle zu erleichtern, wurde bereits in den Abschnitten 2.5 und 3.2.3 angesprochen. Sie nützt aus, daß Hochleistungstransportprotokolle zwar in vielfältigen Varianten auftreten können, die sich jedoch immer aus einer beschränkten Menge von Protokollfunktionen zusammensetzen (siehe Abschnitte 2.2.2 und 2.2.3). Es erscheint deshalb sinnvoll, eine Bibliothek von Spezifikationsbausteinen zu entwerfen. Die Spezifikationsbausteine modellieren einzelne Protokollmechanismen und können durch Parametrisierung an die Eigenschaften spezieller Protokolle (z.B. PDU-Formate) angepaßt werden. Der Protokollentwickler erstellt eine Spezifikation, indem er Bausteine aus der Bibliothek instantiiert und sie dann miteinander komponiert. Eine zweite gleichartige Bibliothek von Kommunikationsdiensteigenschaften ermöglicht ihm auf die gleiche Weise die Erstellung von Dienstspezifikationen. Der in Abschnitt 4.1 vorgestellte kompositionale Spezifikationsstil und die dazu gehörende Spezifikationssprache cTLA$^+$ (siehe Abschnitt 4.3) eignen sich zur Modellierung der Spezifikationsbausteine und ihrer Kompositionen. Die Spezifikationsbausteine werden als cTLA$^+$-Prozeßtypen beschrieben. Der Protokollentwickler kann somit Kompositionen von Spezifikationsbausteinen durch die cTLA$^+$-Prozeßkomposition oder die cTLA$^+$Prozeßkombination spezifizieren.

Außerdem wurde in den Abschnitten 2.5 und 3.2.3 erörtert, daß man nicht nur die Erstellung von Protokoll- und Dienstspezifikationen vereinfachen kann, sondern auch den Beweis, daß ein Protokoll einen bestimmten Dienst erbringt. Das Beweistheorem kann in Lemmata zerlegt werden (siehe Abschnitt 4.2). Jedes Lemma sagt aus, daß eine Diensteigenschaft durch ein Subsystem des Protokolls implementiert wird. Diensteigenschaften werden unabhängig von dem gerade betrachteten Protokoll durch Protokollfunktionen implementiert, die immer aus den gleichen Protokollmechanismen zusammengesetzt sind. Es bietet sich deshalb an, jedes dieser Lemmata einmal zu beweisen und das Beweisergebnis dem Protokollentwickler zur Verfügung zu stellen. Der Entwickler muß selbst nur noch einige Randbedingungen beweisen, um sicherzustellen, daß er die Protokollmechanismen in geeigneter Weise miteinander komponiert hat. Der eigentliche TLA-Verfeinerungsbeweis muß von ihm jedoch nicht mehr erbracht werden.

Wir erstellen somit neben der Spezifikationsbausteinbibliothek auch eine Bibliothek von Theoremen, die Implikationen zwischen Protokollsubsystemspezifikationen und Dienstspezi-

fikationsbausteinen aus den Bausteinbibliotheken beschreiben.

Wie in Abschnitt 4.2 erläutert wurde, muß neben den eigentlichen Verfeinerungsbeweisen noch ein Beweis geführt werden, der sicherstellt, daß die Aktionen eines Feinsystems konsistent zu denen des Grobsystems gekoppelt sind. Beim Entwurf der Spezifikationsbausteine des Frameworks wurde darauf geachtet, daß sinnvollerweise nur Aktionen mit demselben Namen miteinander gekoppelt werden[1]. Außerdem besitzen alle Aktionen einer Grobspezifikation die gleichen Namen wie diejenigen Aktionen der Feinspezifikation, die sie erfüllen. Man kann deshalb sofort nachweisen, daß die Kopplungen des Feinsystems und des Grobsystems zueinander immer konsistent sind, wenn der Nutzer nur Aktionen mit demselben Namen miteinander verknüpft.

In diesem Kapitel soll ein Framework für Hochleistungstransportprotokolle vorgestellt werden, das bereits in [HK95] veröffentlicht wurde. Das Framework besteht aus Bibliotheken von Spezifikationsbausteinen für Protokollmechanismen und Diensteigenschaften sowie aus Bibliotheken bewiesener Theoreme. Der Aufbau des Frameworks, seiner Bibliotheken, der Spezifikationsbausteine und der Theoreme wird in Abschnitt 5.1 näher beschrieben. In den darauf folgenden Abschnitten werden die Spezifikationsbausteine und die Theoreme des Frameworks eingeführt. Zunächst werden in Abschnitt 5.2 die Diensteigenschaften vorgestellt. Die Abschnitte 5.3, 5.4 und 5.5 enthalten die Beschreibungen der Protokollmechanismen und der Basisdienste. Die Theoreme des Frameworks werden in den Abschnitten 5.6 und 5.7 eingeführt. Einige typische Beweise dieser Theoreme werden dort ebenfalls beschrieben.

5.1 Aufbau des Frameworks

Das Framework enthält außer den schon genannten Bibliotheken von Spezifikationsbausteinen und Theoremen einen Abschnitt, in dem die Beweise der Theoreme skizziert sind. Dort sind die Invarianten, Refinement Mappings und die für Lattice-Beweise verwendeten Zustandsfolgen (siehe Abschnitt 3.3.3) angegeben. Das Framework kann über das World Wide Web unter der Adresse

```
http://ls4-www.informatik.uni-dortmund.de/RVS/P-TPM
```

abgerufen werden.

In Abschnitt 5.1.1 erläutern wir die verschiedenen Baustein- und Theorembibliotheken. Aufgrund der Anzahl an Spezifikationsbausteinen und -bibliotheken empfiehlt sich eine Strukturierung der Bausteine und Theoreme abhängig von den Aufgaben, die sie in einem Kommunikationsprotokoll oder -dienst erfüllen. Diese Struktur wird in Abschnitt 5.1.2 erörtert. Der syntaktische und semantische Aufbau der einzelnen Spezifikationsbausteine wird in Abschnitt 5.1.3 und derjenige der Theoreme in Abschnitt 5.1.4 vorgestellt.

5.1.1 Die Bibliotheken des Frameworks

Die Spezifikationen der Diensteigenschaften beschreiben Verhalten wesentlich abstrakter als diejenigen der Protokollmechanismen. Zum einen berücksichtigen sie nicht die im Proto-

[1] Ausnahmen davon bilden nur die Fairnessprozesse, in denen Paare von Aktionen auftreten (siehe Abschnitt 4.3.4). Ihren Aktionsbezeichnern werden zur Unterscheidung die Buchstaben `f` bzw. `n` vorangestellt. Somit werden zum Beispiel die Aktionen `In` der Safetyprozesse mit `fIn` der zugehörigen Fairnessprozesse gekoppelt.

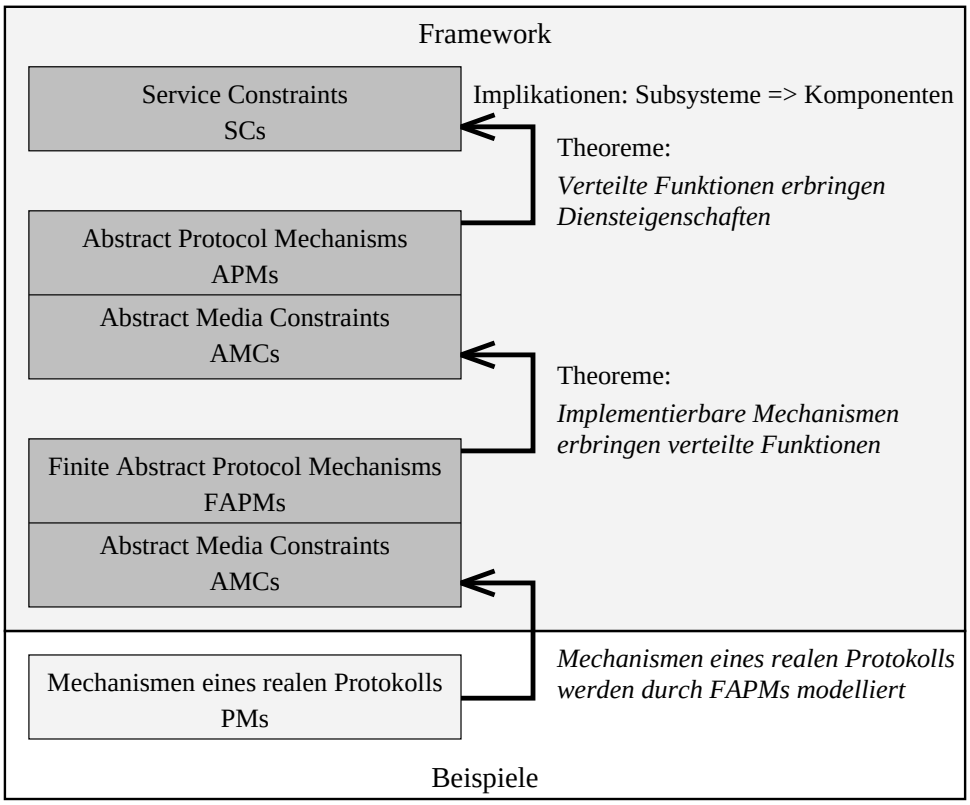

Abbildung 5.1: Aufbau des Frameworks

koll vorhandene räumliche Verteilung der Systemressourcen. Zum anderen kann man, da Diensteigenschaften abstrakte Schnittstellenbeschreibungen sind, die nicht unmittelbar in reale Implementierungen überführt werden, auf die Modellierung implementierungsspezifischer Eigenschaften (z.B. einen endlichen Zustandsraum) verzichten. Um die Theoreme zum Beweis der Diensteigenschaften durch die Protokollmechanismen zu vereinfachen und auch die Größe der Theorembibliotheken zu begrenzen, führen wir (nach dem Prinzip der schrittweisen Verfeinerung) eine weitere Bausteinbibliothek ein. Der Abstraktionsgrad der Spezifikationen dieser Bibliothek liegt dabei zwischen dem von Protokollen und dem von Diensten. Zwar wird die räumliche Verteilung der Protokollkomponenten abgebildet, doch werden Variablen mit unendlich großen Wertebereichen verwendet, so daß Implementierungsdetails wie die Mehrfachnutzung von Sequenznummern nicht berücksichtigt werden müssen. Dadurch kann man die Verfeinerungsbeweise in zwei Schritten erbringen:

1. Beim Beweis, daß Diensteigenschaften durch abstrakte Protokollmechanismen erbracht werden, wird die Korrektheit der Protokollfunktionen überprüft, die die Überbrückung der räumlichen Verteilung realisieren.

2. Bei der Verifikation, daß die abstrakten Protokollmechanismen durch implementierungsnähere Protokollmechanismen erfüllt werden, wird untersucht, ob Übertragungsfehler durch Wiederverwendung von Datensequenznummern und Verbindungskennun-

gen auftreten können.

Das Framework enthält vier Bibliotheken mit Spezifikationsbausteinen (siehe Abbildung 5.1):

- **Service Constraints (SCs):** Die cTLA$^+$-Prozeßtypen dienen zur Spezifikation der Diensteigenschaften von Kommunikationsdiensten.
- **Abstract Media Constraints (AMCs):** Mit diesen cTLA$^+$-Prozeßtypen werden Basisdienste, über die Protokollinstanzen miteinander interagieren[2], spezifiziert. Bis auf einige Ausnahmen (siehe Abschnitt 5.4) entsprechen die AMCs den SCs.
- **Abstract Protocol Mechanisms (APMs):** Mit diesen cTLA$^+$-Prozeßtypen können abstrakte Protokollinstanzen spezifiziert werden, deren Zustandsraum unbegrenzt ist. Eine abstrakte Protokollspezifikation wird durch Komposition von APMs und AMCs erstellt.
- **Finite Abstract Protocol Mechanisms (FAPMs):** Die cTLA$^+$-Prozeßtypen werden zur Spezifikation realer Protokollinstanzen verwendet. Sie besitzen deshalb nur Kontrollstrukturen von einem endlichen Zustandsraum.

Da die Protokollverifikation in zwei Schritten erfolgt, besitzt das Framework zwei Theorembibliotheken:

- **APMs/AMCs nach SCs:** Diese Bibliothek enthält Theoreme für den Beweis, daß einzelne SCs durch abstrakte Protokollfunktionen implementiert werden, die aus einer Komposition von APMs und AMCs bestehen.
- **FAPMs/AMCs nach APMs:** In dieser Bibliothek sind Theoreme für den Beweis zusammengefaßt, daß APMs durch aus FAPMs und AMCs bestehende Subsysteme realer Protokolle erbracht werden.

Skizzen zum Beweis der Theoreme sind jeweils beigefügt.

5.1.2 Die Struktur der Frameworkbibliotheken

Um die Bausteinbibliotheken übersichtlich zu halten, unterteilen wir die Bausteine abhängig von ihren Aufgaben in mehrere Klassen (siehe Abbildung 5.2). Wir berücksichtigen dabei, daß Verbindungen[3] das wesentliche Strukturmerkmal realer Protokolle bilden. Verbindungen sind eigenständige Kommunikationseinheiten, die gegenüber anderen Verbindungen abgeschlossen sind. Zusätzlich kann man Verbindungen gut in die drei Phasen Verbindungsaufbau, Datentransfer und Verbindungsabbau strukturieren. Eine bereits abgebaute Verbindung kann nicht mehr erneut aufgebaut werden [4]. Wir fassen deshalb das Transportsystem eines Rechnernetzes (siehe Abschnitt 2.1) als Menge unendlich vieler Verbindungen auf. Die Koordinierung zwischen mehreren Verbindungen beschränkt sich dabei darauf sicherzustellen, daß die von

[2] Im aktuellen Zustand werden durch das Framework nur Verbindungen zwischen zwei Benutzern modelliert.

[3] Wir definieren verbindungslosen Datentransfer als Folge sehr kurzer Verbindungen und halten uns damit an die Realisierung der Datagrammübertragung in Hochleistungsprotokollen (z.B. XTP, PATROCLOS).

[4] Allerdings kann in realen Protokollen eine Verbindung mit derselben Verbindungskennung wie diejenige einer bereits abgebauten Verbindung aktiv sein.

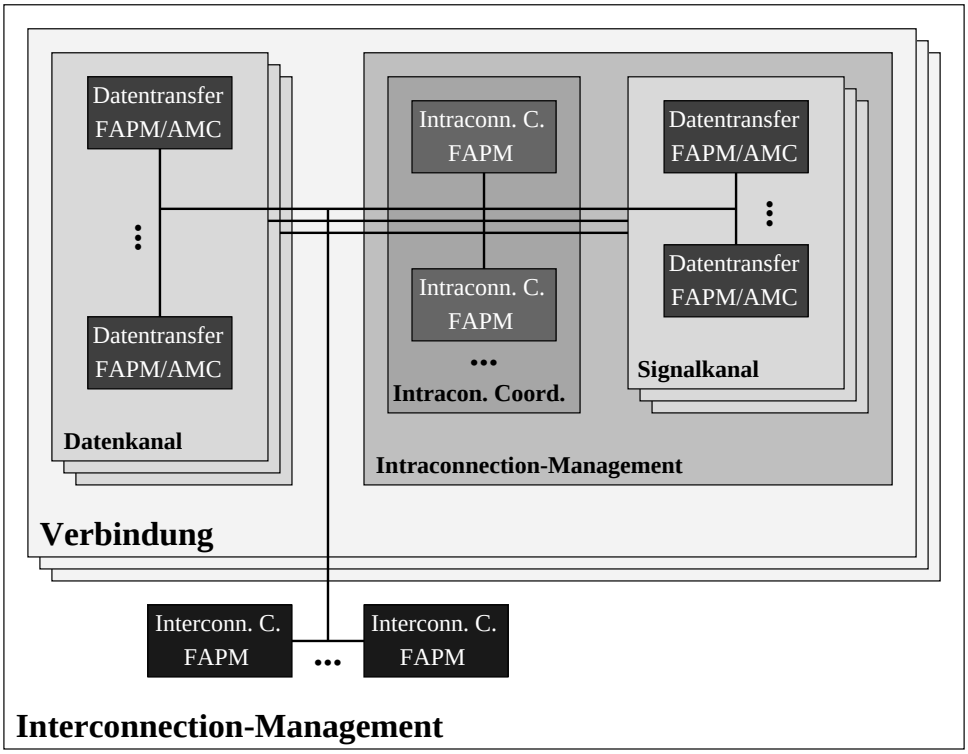

Abbildung 5.2: Komposition von FAPMs und AMCs zu Protokollspezifikationen

einer Station ausgehenden Verbindungen voneinander unterscheidbar sind, und daß jede Station zu einem Zeitpunkt nur eine bestimmte Anzahl offener Verbindungen besitzt. Sie wird durch spezielle Spezifikationsbausteine, den Interconnection-Coordination-Spezifikationen beschrieben.

Eine Verbindung setzt sich aus Komponenten zusammen, die zum einen den Datentransfer und zum anderen die Verbindungsverwaltung (Intraconnection-Management) beschreiben. Die Bausteine des Datentransfers spezifizieren alle Funktionen, die die Übertragung externer Nutzdaten realisieren. Durch Komposition dieser Bausteine (bei Protokollspezifikationen zusammen mit AMCs) wird ein Datenkanal modelliert. Wenn ein Protokoll oder ein Dienst Verbindungen von unterschiedlicher Dienstgüte anbietet, wird für jede mögliche Dienstgüte ein eigener Datentransferkanal gebildet. Dabei kann die Änderung der Dienstgüte innerhalb einer Verbindung durch Wechseln des aktiven Datentransferkanals beschrieben werden. Die Prozesse der Verbindungsverwaltung modellieren die Funktionen zur Koordinierung der Datenkanäle. Sie werden in Komponenten zur Übertragung von Signalisierungsinformation zwischen Dienstzugangspunkten und in die eigentlichen Funktionen der Verbindungskoordinierung unterteilt. Da der Transfer von Signalisierungsdaten dem von Nutzdaten im wesentlichen entspricht, wird er durch die Datentransfer-APMs beschrieben. Da auch die Signalisierungsinformation mit unterschiedlichen Dienstgüten übertragen wird, ist die Definition mehrerer Kanalspezifikationen möglich. Die Funktionen der Koordinierung von Datenkanälen und Signalisierungskanälen werden durch die Intraconnection-Coordination-Spezifikationsbausteine

modelliert. Dazu gehört auch die Beschreibung des von der Systemumgebung (dem Dienstnehmer) vorausgesetzten Verhaltens. Da der kompositionale Spezifikationsstil im Gegensatz zu der von Abadi und Lamport vorgeschlagenen Methode (siehe Abschnitt 3.3.5) keine „Assumption/Guarantee"-Spezifikationen zuläßt, wird das Verhalten der Systemumgebung durch Prozeßspezifikationen modelliert. Abadi und Lamport zeigen in [AL93a], daß diese Kopplungsmethode bei Verfeinerungsbeweisen zu der von ihnen verwendeten Methode der „Assumption/Guarantee"-Systeme äquivalent ist, nach der das erwünschte Systemverhalten nur sichergestellt ist, wenn die Umgebung das von ihr geforderte Verhalten zeigt.

Jede Bausteinbibliothek des Frameworks enthält somit drei verschiedene Klassen:

- Datentransfer (Signalisierungsdatentransfer),
- Intraconnection-Coordination und
- Interconnection-Coordination.

Wir beschreiben nun, wie man instantiierte Spezifikationsbausteine zu Dienst- oder Protokollspezifikationen komponiert. Zuerst werden Bausteine des Datentransfers zu Datenkanälen bzw. Signalisierungskanälen gekoppelt. Bei Duplexverbindungen werden dabei zwei Instanzen jeder Kanalspezifikation verwendet, um die beiden Übertragungsrichtungen zu modellieren. Die Kanalspezifikationen werden anschließend mit den Intraconnection-Coordination-Bausteinen zu einer Verbindungsspezifikation komponiert. Durch die cTLA$^+$-Operation Prozeßfelder (siehe Abschnitt 4.3.2) erzeugt man aus einer Verbindungsspezifikation ein Feld beliebig vieler voneinander unabhängiger Verbindungen. Die endgültige Protokoll- oder Dienstspezifikation wird erstellt, indem man das Feld der Verbindungsspezifikationen mit den Interconnection-Coordination-Bausteinen komponiert.

5.1.3 Der Aufbau der Spezifikationsbausteine

Die Spezifikationsbausteine des Datentransfers beschreiben Aspekte der Datenübertragung zwischen räumlich verteilten Rechnerknoten. Sie werden zu Spezifikationen von Simplex-Datenkanälen komponiert. Die Abbildung 5.3 zeigt den Aufbau dieser Datenkanäle bei Dienstspezifikationen (linke Seite) und bei Protokollspezifikationen (rechte Seite). Die Übergabe von Daten an der Transportdienstschnittstelle zur Übertragung zu einem Kommunikationspartner (Dienstprimitiv *T-Data-Request*) wird durch die Aktion `Rq` spezifiziert. Die Aktion `In` modelliert die Auslieferung übertragener Daten an den empfangenden Dienstnehmer (Dienstprimitiv *T-Data-Indication*). Durch die Aktion `Cf` wird die implizite Bestätigung[5] ausgelieferter Daten bei deren Sender beschrieben (Dienstprimitiv *T-Data-Confirmation*).

In den Protokollspezifikationen werden vier Systemressourcen modelliert. Die Protokollmechanismen der Sendestation sind zu der Protokollinstanz S und diejenigen des Empfängers zu der Protokollinstanz R zusammengefaßt. Die Spezifikationen dieser Instanzen werden durch Komposition von APMs bzw. FAPMs erzeugt. Die Instanzen kooperieren miteinander durch Transfer von Protokolldateneinheiten (PDUs) über ein untergeordnetes Medium (Dienst), das durch die Simplex-Basiskanäle M und A modelliert wird. Über M werden die Nutzdaten und Protokollsteuerinformation (PCI) von Instanz S zur Instanz R übermittelt,

[5]Explizite Bestätigungen von Diensten werden durch einen eigenen Datenkanal modelliert, in dem `Rq` für das Dienstprimitiv *T-Data-Response* und `In` für *T-Data-Confirmation* verwendet wird.

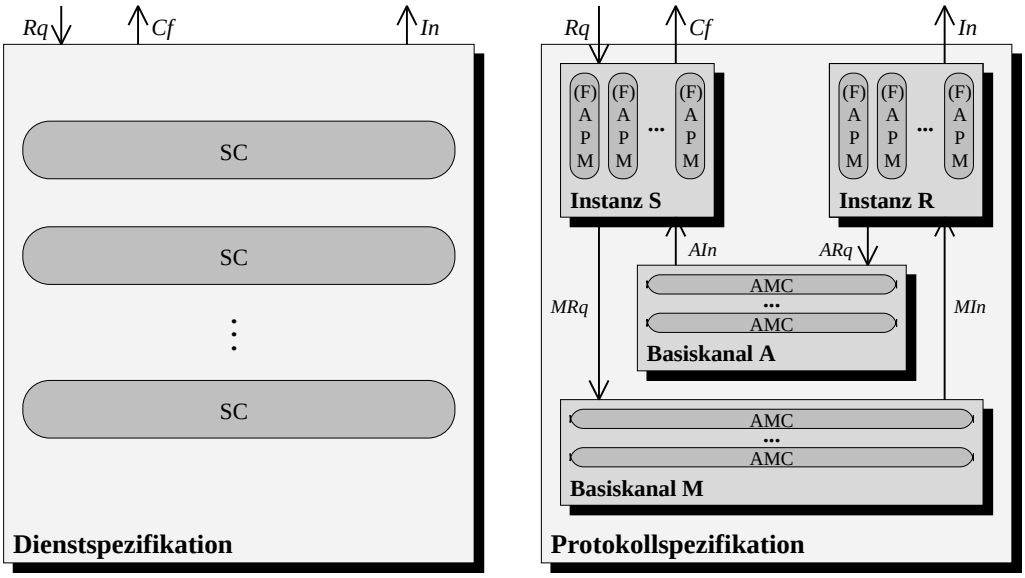

Abbildung 5.3: Aufbau der Datentransferkanäle

während über A PCI von R nach S gesendet wird. Die Kanäle M und A werden durch AMCs beschrieben. Das Verhalten an der Dienstschnittstelle wird durch die Aktionen `MRq`, `MIn`, `ARq` und `AIn` spezifiziert. Die Aktionen `MRq` und `MIn` modellieren das Senden bzw. Empfangen von Daten über Kanal M. Analog dazu beschreiben die Aktionen `ARq` und `AIn` den Zugriff auf Kanal A.

AMCs werden auch zur Spezifikation von Signalisierungskanälen verwendet. Zur besseren Unterscheidung fügen wir den Aktionen in diesen Kanälen den Buchstaben `S` (Signal) als Präfix hinzu (z.B. `SMRq`). Spezielle Signalisierungskanäle sind zur Beschreibung von Verbindungsabbrüchen notwendig. Im Gegensatz zu allen anderen Protokollfunktionen oder Diensteigenschaften werden Verbindungsabbrüche nicht zwingend von einem Dienstnehmer ausgelöst, sondern können auch vom Diensterbringer initiiert werden. Wir zeigen in Abschnitt 5.2.3, daß man dennoch normale Signalisierungskanäle zur Beschreibung von Verbindungsabbrüchen nutzen kann. Zur besseren Unterscheidung ersetzen wir den Bezeichner `SRq` in diesem Fall durch `PAbI` (Dienstprimitiv *Provider-Abort (Initiator)*) und `SIn` durch `PAbP` (Dienstprimitiv *Provider-Abort (Peer)*).

Die Kooperation der verschiedenen Kanäle einer Verbindung wird durch die Spezifikationsbausteine der Intraconnection-Coordination modelliert. Die meisten dieser Spezifikationen sind genau einer Protokollinstanz zugeordnet. Sie verwenden die Aktionen `Rq`, `In`, `Cf`, `SRq`, `SIn`, `SCf`, `PAbI` und `PAbP`, die mit den gleichnamigen Aktionen der Datenkanalspezifikationen gekoppelt werden.

Die Spezifikationsbausteine der Interconnection-Coordination modellieren die Kooperation zwischen einzelnen Verbindungen. Sie sind ebenfalls einer lokalen Instanz zugeordnet und verwenden Aktionen `SRq` und `SIn`, die mit den gleich bezeichneten Aktionen der Kanalspezifikationen und der Intraconnection-Coordination gekoppelt werden.

5.1.4 Die Syntax der Theoreme

Die Theoreme des Frameworks beschreiben, daß ein Prozeß eines abstrakten Grobsystems (z.B. ein SC) durch ein Subsystem aus Bausteinen eines detaillierteren Feinsystems (z.B. APMs und AMCs) implementiert wird. Ein Theorem zum Beweis des SCs `LiveIn`, das in Abschnitt 5.2.2 vorgestellt wird, ist im folgenden aufgeführt:

```
LET Pars ≜  Anforderungen an die Prozesstypparameter
            mla = { (p,q) | skey[spci[p] ] = skey[spci[q] ] ∧
                            stack[spci[p] ] = stack[spci[q] ] ∧
                            stcre[spci[p] ] = stcre[spci[q] ] ∧
                            p ∈ encpdu ⇔ q ∈ encpdu } ∧
            mtc ⊆ { (p, q) | q ∉ encpdu ∨ spci[p] = spci[q] };
            ala = { (p,q) | sack[spci[p] ] = sack[spci[q] ] ∧
                            snak[spci[p] ] = snak[spci[q] ] ∧
                            scre[spci[p] ] = scre[spci[q] ] ∧
                            p ∈ encpdu ⇔ q ∈ encpdu } ∧
            atc ⊆ { (p, q) | q ∉ encpdu ∨ spci[p] = spci[q] };
    Sys ≜   Subsystem aus APMs und AMCs, durch das LiveIn erfuellt wird
            SLiveMRq (pdu, pci, usd, encpdu, spci, skey, sack, snak, scre,
                      stack, stcre, skk, skn, skm, usdsize, usdsplit, mcr, rcc) ∧
            RLiveARq (pdu, pci, usd, encpdu, spci, skey, sack, snak, scre,
                      stack, stcre, skk, skn, skm, usdsize, rcu, rcc, ma, mr, mo) ∧
            RLiveIn (usd, skk, skn, skm, usdsize, usdsplit, rcu, rcc, tg) ∧
            RAcknowledge (pdu, pci, encpdu, spci, sack, skk, rcu, rcc, ma) ∧
            MSDUId ∧
            MCorruptions (pdu, mtc) ∧
            MPhantoms (pdu, encpdu) ∧
            MLiveIn (pdu, mmaxrg, mtg, mc, mla) ∧
            ASDUId ∧
            ACorruptions (pdu, atc) ∧
            APhantoms (pdu, encpdu) ∧
            ALiveIn (pdu, amaxrg, atg, ac, ala) ∧
            CC_LiveIn;    Kopplungsbeschreibung von Sys
    EnvCond ≜  Umgebung toleriert faire Aktionen von Sys?
            ∀ krq,p,kd : Enabled(SLiveMRq.fMRq(krq,p,kd)) ⇒
                                                (krq,p,kd) ∈ Sys.e_fMRq ∧
            ∀ p,kd : Enabled(RLiveARq.fARq(p,kd)) ⇒ (p,kd) ∈ Sys.e_fARq ∧
            ∀ krq,d : Enabled(RLiveIn.fIn(krq,d)) ⇒ (krq,d) ∈ Sys.e_fIn ∧
            ∀ krq : Enabled(MLiveIn.fMIn(krq)) ⇒ krq ∈ Sys.e_fMIn ∧
            ∀ d : Enabled(MLiveIn.fMnoTick(d)) ⇒ d ∈ Sys.e_fMnoTick ∧
            ∀ krq : Enabled(ALiveIn.fAIn(krq)) ⇒ krq ∈ Sys.e_fAIn;
            ∀ d : Enabled(ALiveIn.fANoTick(d)) ⇒ d ∈ Sys.e_fANoTick;
IN Pars ∧ Sys ∧ □ EnvCond ⇒ LiveIn (usd, 0, tg, c, ∅)    Theorem
```

Zur Erlangung einer übersichtlichen Darstellung wird die von TLA$^+$ verwendete Schreibweise `LET` $\langle Definitionen \rangle$ `IN` $\langle Theorem \rangle$ benutzt. Das eigentliche Theorem ist hinter dem Schlüsselwort `IN` aufgeführt. Das SC *LiveIn* wird durch ein Subsystem `Sys` des Feinsystems erbracht, wenn die Nebenbedingungen `Pars` und `□EnvCond` gelten.

Das Subsystem `Sys` und die Nebenbedingungen sind in dem von `LET` angeführten Bereich enthalten. Die Randbedingung `Pars` enthält Anforderungen an die von den Prozessen in `Sys`

und dem Grobsystem `LiveIn` verwendeten Prozeßparametern. In `Sys` sind die Prozeßtypen des Feinsystems beschrieben, aus denen der Prozeßtyp gebildet wird, der das Subsystem modelliert. Die Kopplung der Aktionen wird durch die Kopplungsbeschreibung (z.B. $\texttt{CC}_{\texttt{LiveIn}}$) spezifiziert, die aus Gründen der Übersichtlichkeit nicht explizit aufgeführt ist. Da immer alle Aktionen aus den Prozeßtypen von `Sys`, die denselben Namen haben, miteinander gekoppelt werden, kann man die Kopplungsbeschreibung einfach herleiten. Ausnahmefälle werden in den Theoremen gesondert dokumentiert.

Das Feld `EnvCond` ist nur in Theoremen zum Beweis von Livenessprozeßtypen enthalten. In Abschnitt 4.3.4 wurde eine Methode zur Strukturierung von Fairnessbeweisen erläutert. Wesentliches Merkmal dieser Methode ist die Zerlegung der zu komponierenden Prozesse in Safety- und Livenessprozesse und die Anforderung, daß eine faire Aktion nur schaltbar sein darf, wenn alle mit ihr gekoppelten Aktionen aus Safetyprozessen auch schaltbar sind. Beim Beweis des Theorems wurde diese Eigenschaft für die in `Sys` enthaltenen Prozeßtypen verifiziert. Durch `EnvCond` wird sie auch für andere Safetyprozesse sichergestellt, die zwar zum Feinsystem, aber nicht zu `Sys` gehören. Der Beweis von $\Box$`EnvCond` wird erleichtert, da in der Theoremdokumentation eine Liste von Spezifikationsbausteinen enthalten ist, für die die Gültigkeit von $\Box$`EnvCond` bereits bewiesen wurde.

Verfeinerungsbeweise werden unter Verwendung von Theoremen erbracht, indem man ihre Randbedingungen `Pars` und $\Box$ `EnvCond` nachweist. Außerdem muß noch sichergestellt werden, daß die Feinsystemspezifikation alle Prozeßtypen aus `Sys` enthält.

5.2 Service Constraints (SCs)

Dienstspezifikationen beschreiben das Verhalten der Kommunikationsdienste, ohne auf Verteilungsaspekte einzugehen. Sie setzen sich aus Diensteigenschaften zusammen, die wir im Framework durch die SCs modellieren. Die SCs werden im folgenden vorgestellt. In den Abschnitten 5.2.1 und 5.2.2 führen wir zunächst die Beschreibungen von Safety- bzw. Livenesseigenschaften des Datentransfers ein. Danach beschreiben wir in Abschnitt 5.2.3, wie die SCs des Datentransfers für die Spezifikation von Signalkanälen genutzt werden. Die SCs der Intraconnection-Coordination werden in den Abschnitten 5.2.4 und 5.2.5 erläutert. In Abschnitt 5.2.6 gehen wir auf die SCs der Interconnection-Coordination ein.

5.2.1 Safety-SCs des Datentransfers

Wichtige funktionale Eigenschaften eines Datenkanals sind die Arten an Datenübertragungsfehlern, die vom Dienst toleriert bzw. ausgeschlossen werden. Man unterscheidet fünf verschiedene Fehlerarten: Datenverfälschungen, duplizierte Daten, Datenvertauschungen, Datenverluste und Phantome (vgl. [LKK93]). Durch die SCs kann man Fehler einer dieser Arten ausschließen oder zumindest einschränken. Dabei können Fehler absolut oder relativ eingeschränkt werden. Ein Beispiel einer absoluten Einschränkung ist die Eigenschaft, daß zwischen zwei korrekt empfangenen Dateneinheiten nie mehr als zwei verlorengegangene Dateneinheiten liegen dürfen. Dagegen bedeutet die Eigenschaft eines Systems, daß höchstens 5 % aller übertragenen Dateneinheiten verlorengehen dürfen, eine relative Eigenschaft. Für die Fehlerarten der Datenverfälschungen, der Datenverluste und der Phantome sind jeweils zwei SCs im Framework vorhanden, so daß man absolute und relative Einschränkungen modellieren

kann. Die anderen Fehlerklassen werden dagegen nur durch jeweils ein SC mit absoluter Einschränkung beschrieben, da entsprechende SCs mit relativen Einschränkungen nicht sinnvoll durch Protokollmechanismen realisiert werden können. Außer Übertragungsfehlern modelliert man durch SCs auch eine begrenzte Kanalkapazität, die Anzahl gesendeter, aber noch nicht ausgelieferter Daten, und die in Hochleistungstransportprotokollen gerne verwendete Methode impliziter Bestätigungen.

Im folgenden skizzieren wir den Aufbau der SCs. In Abschnitt 5.1.3 wurde beschrieben, daß die Sendung von Nutzdaten durch die Aktion `Rq`, die Auslieferung durch die Aktion `In` und implizite Bestätigungen durch die Aktion `Cf` modelliert werden. Zur Beschreibung der übertragbaren Dateneinheiten besitzen die SCs den Datentyp `usd` als Prozeßtypparameter. Bei der Modellierung von byteweisem Datentransfer instantiiert der Spezifizierer zum Beispiel `usd` mit der Menge $\{0, \ldots, 255\}$. Datenströme werden jedoch nicht nur durch die Werte der Daten sondern auch durch ihre Sendereihenfolge beschrieben. Wir führen deshalb einen weiteren Datentyp `key` ein, der aus den natürlichen Zahlen und einem weiteren Wert `'notsent'` besteht. Bei der Sendung wird jeder Dateneinheit eine Sequenznummer in aufsteigender Reihenfolge zugewiesen, so daß eine Ordnung aller gesendeten Daten modelliert wird. Phantome sind durch `'notsent'` gekennzeichnet. Eine Dateneinheit wird somit während der Übertragung durch ein geordnetes Paar aus der Menge (`key` $\times$ `usd`) identifiziert. In den Aktionen `Rq` und `In` wird die Sequenznummer der bearbeiteten Dateneinheit durch den Aktionsparameter `krq` und der Inhalt durch `d` referenziert. `Cf` verwendet nur `krq`.

Bevor wir auf die SCs zur Modellierung von Übertragungsfehlern eingehen, stellen wir zunächst das SC *SDUId* vor, das als Basisbaustein in jeder Dienstspezifikation verwendet wird. Es dient zur Spezifikation der Sequenznummernvergabe:

```
PROCESS SDUId    Sequenznummernvergabe; alle Phantome sind erkennbar
IMPORT Constants;
BODY
  VARIABLES
    cRq : key;    Sequenznummernzaehler
  INIT ≙ cRq = 0 ;
  ACTIONS
    Rq (krq : key) ≙ Uebergabe vom Dienstnehmer
      krq = cRq ∧ cRq′ = cRq + 1 ;
    In (krq : key) ≙ Auslieferung an den Dienstnehmer
      ( (krq = "notsent") ∨ (krq < cRq) ) ∧ cRq′ = cRq ;
END
```

Neu gesendeten Daten (Aktion `Rq`) werden natürliche Zahlen in aufsteigender Reihenfolge als Sequenznummern zugewiesen. Dazu speichert eine Variable `cRq` die als nächstes zu vergebende Sequenznummer. `cRq` wird bei jeder Schaltung von `Rq` inkrementiert. Man sieht sofort, daß immer nur Daten gesendet wurden, deren Sequenznummer kleiner als der aktuelle Wert des Zählers `cRq` ist. Alle Daten, die an den Empfänger ausgeliefert werden, besitzen also entweder Sequenznummern, die kleiner als `cRq` sind, oder sind Phantome. Daraus folgt die Schaltbedingung der Aktion `In`, nach der alle Phantome durch die spezielle Sequenznummer `'notsent'` gekennzeichnet sind.

Wir betrachten im folgenden die SCs zur Modellierung von Übertragungsfehlern. Durch das SC *Corruptions* wird beschrieben, ob und bis zu welchem Grad Nutzdaten verfälscht werden dürfen:

```
PROCESS Corruptions    Sicherung gegen Datenverfaelschungen
                     ( usd : Any CONSTANT;                          Nutzdaten
                       tc :  SUBSET(usd × usd) CONSTANT )  Menge erlaubter Verfaelschungen
IMPORT Constants;
BODY
  VARIABLES
    buf : SUBSET(key × usd);     Menge gesendeter Daten
  INIT ≙ buf = ∅;
  ACTIONS
    Rq (krq : key; d : usd) ≙  Uebergabe vom Dienstnehmer
      buf′ = buf ∪ {(krq,d)} ;
    In (krq : key; d : usd) ≙  Auslieferung an den Dienstnehmer
      ( krq = "notsent" ∨ ∀ e ∈ usd :: ((krq,e) ∉ buf) ∨
        ∃ e ∈ usd :: ((krq,e) ∈ buf ∧ (e,d) ∈ tc ) ) ∧
      buf′ = buf ;
  END
```

Das SC hat neben `usd` einen weiteren Prozeßtypparameter `tc`, der die Menge tolerierbarer Datenverfälschungen angibt[6]. Wenn `tc` das Element `(x,y)` enthält, darf anstatt einer mit der Sequenznummer `k` gesendeten Dateneinheit `(k,x)` auch die Dateneinheit `(k,y)` ausgeliefert werden (`tc`$\triangleq\{(x,x)|x \in$ `usd`$\}$ verbietet Verfälschungen). *Corruptions* schränkt Datenverfälschungen also absolut ein. Alle bisher gesendeten Dateneinheiten werden in der Mengenvariable `buf` gespeichert. Durch die Aktion `In` können außer Phantomen nur von der Relation `tc` tolerierte Dateneinheiten ausgeliefert werden (∃ `e` ∈ `usd` :: `((krq,e)` ∈ `buf` ∧ `(e,d)` ∈ `tc )`).

Die anderen Datentypen werden hier nur kurz skizziert. Eine ausführliche Beschreibung einschließlich der Spezifikationen findet sich in der Frameworkbeschreibung im World Wide Web.

Das SC *RelativeCorruptions* erlaubt die Beschreibung eines relativen Anteils an verfälschten Daten im Vergleich zu der Größe des gesamten Datenstroms. Außer `tcr`, das dem Parameter `tc` in SC *Corruptions* entspricht, und `usd` stehen zwei weitere Prozeßtypparameter zur Verfügung. `maxrc` gibt die relative Häufigkeit verfälschter Daten an, die vom Dienstnehmer toleriert wird. Durch `usdsize` kann man den Dateneinheiten eine Datenlänge zuweisen, die bei der Berechnung des aktuellen Anteils verfälschter Daten berücksichtigt wird. Das SC schränkt die Auslieferung von Daten ein, die weder Phantome sind noch zu den gesendeten Daten in der Relation `tcr` stehen. Verfälschte Daten dürfen nur ausgeliefert werden, wenn dadurch das in `maxrc` beschriebene Limit nicht überschritten wird.

Das SC *Duplicates* besitzt einen Prozeßtypparameter `td`, durch den die maximale Anzahl an Wiederholungen angegeben wird (`td` $= 0$ verbietet Wiederholungen).

Das SC *Reorderings* beschränkt Vertauschungen im Ausgangsdatenstrom und verhindert die Auslieferung veralteter Nutzdaten. Es verwendet eine Variable `maxIn`, die den um 1 inkrementierten Wert der größten aller bisher ausgelieferten Sequenznummern speichert. `maxIn` wird auch von den im folgenden skizzierten SCs *Gaps*, *RelativeGaps* und *Capacity* eingesetzt. In *Reorderings* legt der Prozeßtypparameter `tr` fest, daß nur Dateneinheiten ausgeliefert werden dürfen, deren Sequenznummer größer oder gleich `maxIn - tr - 1` ist.

Die Beschreibung der Limitierung von Datenverlusten ist komplizierter als diejenige der anderen Fehlerarten, da sie sowohl Safety- als auch Livenessannahmen betrifft. Ursache eines

[6] `tc` muß eine Obermenge von $\{(x,x)|x \in$ `usd`$\}$ sein.

Datenverlustes kann zum einen eine Datenvertauschung sein. Das Auftreten einer Lücke im Ausgabedatenstrom kann nämlich dazu führen, daß die Auslieferung der Daten innerhalb dieser Lücke durch das SC *Reorderings* ausgeschlossen wird, so daß sie verlorengehen. Zum anderen kann aber auch fehlende Liveness des Dienstes ein Grund für Datenverluste sein. Bleibt der Dienst im Verlauf einer Datenübertragung in irgendeinem Zustand für immer stehen, in dem Daten zwar gesendet aber noch nicht ausgeliefert wurden, sind diese Daten aus der Sicht des Dienstnehmers ebenfalls verloren. Im Framework modellieren wir durch das SC *Gaps* die Safetyeigenschaft, daß Lücken im Ausgabestrom eine bestimmte Länge nicht überschreiten dürfen. Dazu dient der Parameter `tg`, der festlegt, daß eine Dateneinheit nur ausgeliefert werden darf, wenn ihre Kennung höchstens um `tg` größer als `maxIn` ist. (`tg` = 0 schließt Lücken im Datenstrom ganz aus.) Die Livenessaspekte werden durch die SCs *LiveIn* und *LiveInNoAttr* modelliert, die wir im folgenden Abschnitt vorstellen. Durch das SC *RelativeGaps* wird spezifiziert, daß der Anteil an Lücken im Datenstrom im Vergleich zum gesamten Datenstrom nicht den in Prozeßtypparameter `maxrg` gegebenen relativen Wert übersteigen darf.

Mit Hilfe des SC *Phantoms* kann die Auslieferung von Phantomen eingeschränkt oder ganz ausgeschlossen werden. Der Prozeßtypparameter `pd` gibt die Menge an Nutzdaten an, die nicht ausgeliefert werden dürfen, wenn ihre Sequenznummer den Wert `'notsent'` hat. Im Gegensatz dazu beschreibt das SC *RelativePhantoms* den tolerierbaren relativen Anteil an Phantomen im Ausgabedatenstrom, der durch den Prozeßtypparameter `maxrp` angegeben wird.

Die Kapazität des Dienstes wird durch das SC *Capacity* beschrieben. Es besitzt einen Prozeßtypparameter `c`. Die Sendung einer neuen Dateneinheit (Aktion `Rq`) wird nur erlaubt, wenn deren Sequenznummer um höchstens `c` größer als `maxIn` ist. Es wird dabei davon ausgegangen, daß alle Daten mit Kennungen, die kleiner als `maxIn` sind, veraltet sind und bei der Kapazitätsangabe nicht berücksichtigt werden müssen.

Durch das SC *ShowDelivery* modelliert man die implizite Bestätigung ausgelieferter Dateneinheiten bei ihrem Sender. Eine Dateneinheit, deren Sequenznummer kleiner als `maxIn` ist, darf genau einmal beim Sender bestätigt werden. Dazu dient die Aktion `Cf`. Dieses SC wird im Normalfall nicht zur Spezifikation von Datentransferkanälen, sondern von Signalisierungskanälen eingesetzt (siehe Abschnitt 5.2.3).

5.2.2 Liveness-SCs des Datentransfers

In Transportdiensten sind drei Livenesseigenschaften von Bedeutung. Zum einen muß garantiert sein, daß gesendete Dateneinheiten irgendwann ausgeliefert werden. Zum zweiten darf die Sendung neuer Dateneinheiten nicht permanent aufgrund von Kapazitätsengpässen des Dienstes (Safety-SC *Capacity*) gesperrt sein. Zum dritten muß bei Verwendung impliziter Bestätigungen gewährleistet sein, daß empfangene Dateneinheiten irgendwann bestätigt werden. Wir stellen im folgenden drei SCs vor, durch die diese Anforderungen erfüllt werden. Die SCs *LiveInNoAttr* und *LiveIn* modellieren die ersten beiden Livenesseigenschaften, während die dritte Livenesseigenschaft durch das SC *LiveCf* beschrieben wird. Zunächst führen wir das SC *LiveInNoAttr* ein:

```
PROCESS LiveInNoAttr    Liveness des Dienstes
IMPORT Constants;
BODY
VARIABLES
```

```
    cRq : key ;          Sequenznummernzaehler
    maxIn : key ;        Hoechste Sequenznummer der ausgelieferten Daten
INIT ≙ cRq = 0 ∧ maxIn = 0 ;
ACTIONS
    Rq ≙                 Uebergabe vom Dienstnehmer
      cRq' = cRq + 1 ∧ maxIn' = maxIn;
    fIn ( krq : key ) ≙  Faire Auslieferung an Dienstnehmer
      krq ≠ "notsent" ∧ krq = maxIn ∧ maxIn < cRq ∧
      maxIn' = maxIn + 1 ∧ cRq' = cRq;
    nIn ( krq : key ) ≙  Nicht faire Auslieferung an Dienstnehmer
      not (krq ≠ "notsent" ∧ krq = maxIn ∧ maxIn < cRq) ∧
      maxIn' = IF (krq = "notsent") THEN maxIn ELSE max(maxIn,krq + 1) ∧
      cRq' = cRq;
  WF: fIn;   Fairnessanforderung
END
```

Das SC erfüllt die in Abschnitt 4.3.4 angegebenen Bedingungen an Liveness-SCs. Die Aktion `Rq` ist immer schaltbar. Die Aktion `In` wurde in zwei Aktionen `fIn` und `nIn` mit zueinander komplementären Schaltbedingungen, aber gleichen Zustandsübergängen[7] zerlegt.

Die Variablen `cRq` und `maxIn` haben dieselbe Bedeutung wie in den Safety-SCs. `cRq` zeigt die Sequenznummer an, die der nächsten gesendeten Dateneinheit zugewiesen wird. `maxIn` beschreibt den inkrementierten Wert der höchsten Sequenznummer aller bisher ausgelieferten Dateneinheiten. In einem reihenfolgetreuen Datenstrom wird immer als nächstes die Dateneinheit ausgeliefert, deren Sequenznummer mit `maxIn` identisch ist. Die schwache Fairness der Aktion `fIn` sichert zu, daß eine Dateneinheit mit der Sequenznummer `maxIn` (oder einer größeren Sequenznummer) irgendwann ausgeliefert wird, wenn sie zuvor gesendet wurde (`maxIn` $<$ `cRq`). Dadurch ist sichergestellt, daß mit Ausnahme von tolerierbaren Lücken im Ausgabedatenstrom alle gesendeten Dateneinheiten irgendwann ausgeliefert werden. Durch *LiveInNoAttr* wird auch die zweite Livenessanforderung garantiert, daß irgendwann neue Dateneinheiten gesendet werden dürfen. Das Safety-SC *Capacity* läßt nämlich immer die Sendung mindestens einer neuen Dateneinheit zu, wenn zuvor eine Dateneinheit mit einer Sequenznummer größer oder gleich `maxIn` ausgeliefert wurde.

LiveInNoAttr ist zu allen Safety-SCs des Datentransfers (siehe vorhergehenden Abschnitt) kompatibel, da die Schaltbedingungen der Aktion `In` dieser SCs durch die Schaltbedingung der fairen Aktion `fIn` impliziert werden. Dagegen wird die Schaltbedingung der Aktion `In` des Intraconnection-Coordination-SCs *DataChanOpenR* (siehe Abschnitt 5.2.4), das die Verhinderung von Datenauslieferungen außerhalb der Datentransferphase modelliert, nicht durch `fIn` impliziert. Dadurch wird ausgedrückt, daß garantierter Fortschritt bei der Auslieferung von Daten nur sichergestellt ist, wenn eine Verbindung erst nach der Auslieferung aller Dateneinheiten abgebaut wird.

Als zweites Liveness-SC stellen wir nun *LiveIn* vor, das die Auslieferung einer Dateneinheit aus einer bestimmten Gruppe von Daten garantiert, wenn immer wieder Daten aus dieser Gruppe gesendet werden. Damit kann man die Möglichkeit ausschließen, daß in verlustbehafteten Diensten für den Kommunikationsfortschritt relevante Dateneinheiten immer verlorengehen und nur für die Kommunikation redundante Daten ausgeliefert werden:

[7] Die Beschreibung des Zustandsübergangs der Variablen `maxIn` in Aktion `fIn` ist lediglich eine Vereinfachung desjenigen aus Aktion `nIn`.

```
PROCESS LiveIn    Liveness des Diensten mit der Garantie der Auslieferung von "wichtigen" Daten
                ( usd : Any;                          Nutzdaten
                  maxrg : Real ;                      Anteil erlaubter Luecken
                  tg : Nat ∪ {"infty"} ;              Maximale Groesse von Luecken
                  c : Nat ∪ {"infty"} ;               Dienstkapazitaet
                  lrel : SUBSET(usd × usd) )          Menge der zu einer Gruppe gehoerenden Daten
IMPORT Constants;
BODY
  VARIABLES
    buf : SUBSET(key × usd);      Menge aller gesendeten Daten
    maxIn : key;                  Hoechste Sequenznummer der ausgelieferten Daten
    maxTick : key;                Naechste auszuliefernde Dateneinheit
    ngaps : key;                  Gesamtanzahl aufgetretener Luecken
    stdel : {"no", "yes"};        Darf eine Luecke auftreten ?
  INIT ≜ buf = ∅ ∧ maxIn = 0 ∧ maxTick = 0 ∧ ngaps = 0 ∧
         stdel = "no";
  ACTIONS
    Rq (krq : key; d : usd) ≜     Uebergabe vom Dienstnehmer
      buf′ = buf ∪ {(krq,d)} ∧
      maxIn′ = maxIn ∧ maxTick′ = maxTick ∧ ngaps′ = ngaps ∧
      stdel′ = stdel;
    fIn ( krq : key) ≜            Faire Auslieferung an Dienstnehmer
      krq = maxTick ∧ ∃ d ∈ usd :: (krq, d) ∈ buf ∧
      maxIn′ = maxTick + 1 ∧
      maxTick′ = maxTick + 1 ∧
      ngaps′ = ngaps + maxTick - maxIn ∧
      buf′ = buf ∧ stdel′ = "no";
    nIn ( krq : key ) ≜           Nicht faire Auslieferung an Dienstnehmer
      not( krq = maxTick ∧ ∃ d ∈ usd :: (krq, d) ∈ buf ) ∧
      krq ≤ maxTick ∧
      maxIn′ = IF (krq = "notsent")
               THEN maxIn ELSE max(maxIn, krq + 1) ∧
      maxTick′ = IF (krq = "notsent")
                 THEN maxTick ELSE max(maxTick, krq + 1) ∧
      ngaps′ = IF (krq = "notsent")
               THEN ngaps ELSE ngaps + max(0, krq - maxIn) ∧
      buf′ = buf ∧ stdel′ = IF (krq = maxTick) THEN "no" ELSE stdel;
    Tick ≜                        Enstehung einer Luecke
      maxTick′ = IF ( ∃ d ∈ usd :: (maxTick, d) ∈ buf ∧
                      stdel = "no" ∧
                      ((ngaps + maxTick - maxIn + 1) / (maxTick + 2))
                        ≤ maxrg ∧
                      (maxTick < maxIn + tg ∨ tg = "infty") ∧
                      (maxTick < maxIn + c - 1 ∨ c = "infty") )
                 THEN maxTick + 1 ELSE maxTick ∧
      ngaps′ = ngaps ∧
      buf′ = buf ∧ maxIn′ = maxIn ∧ stdel′ = stdel;
    fNoTick (d : usd) ≜           Verbot von Luecken
      ∃ e ∈ usd :: (maxTick,e) ∈ buf ∧ (e,d) ∈ lrel ∧
      stdel = "no" ∧
      ((ngaps + maxTick - maxIn + 1) / (maxTick + 2)) ≤ maxrg ∧
```

```
      (maxTick < maxIn + tg  ∨  tg = "infty") ∧
      (maxTick < maxIn + c - 1  ∨  c = "infty") ∧
      stdel′ = "yes" ∧
      maxTick′ = maxTick  ∧  maxIn′ = maxIn  ∧  buf′ = buf ∧
      ngaps′ = ngaps;
  SF: fNoTick;   Fairnessanforderungen
  WF: fIn;
END
```

Die Aktionen `Rq`, `fIn` und `nIn` entsprechen im wesentlichen denen in SC *LiveInNoAttr*. Allerdings hängt die Sequenznummer der durch `fIn` ausgelieferten Dateneinheit nicht von `maxIn`, sondern von einer Variablen `maxTick` ab, deren aktueller Wert immer größer oder gleich demjenigen von `maxIn` ist. Dadurch wird modelliert, daß die Nutzdaten mit Sequenznummern zwischen `maxIn` und `maxTick - 1` in einer Lücke liegen und verlorengehen können. Da das Liveness-SC keinen Einfluß auf die Safety des modellierten Systems haben darf, erlaubt die Aktion `nIn` auch die Auslieferung von Dateneinheiten mit Sequenznummern größer als `maxTick`. Das Auftreten eines potentiellen Datenverlustes wird durch die Aktion `Tick` spezifiziert. Diese Aktion beschreibt, daß `maxTick` erhöht und damit die Lücke vergrößert wird, wenn dadurch weder die erlaubte absolute noch die relative Obergrenze von Lücken im Ausgabestrom überschritten werden. Außerdem darf die Lücke nicht größer als die Gesamtkapazität des Dienstes sein, da sonst eine Verklemmung auftreten kann.

Die Eigenschaft, daß bei regelmäßiger Sendung von Daten aus einer bestimmten Gruppe die Auslieferung einer dieser Dateneinheiten sichergestellt ist, wird durch die Aktion `fNoTick` modelliert. Die Relation `lrel` gibt die Gruppeneinteilung an. Wenn der Sender regelmäßig Daten aus einer Gruppe sendet, entspricht `maxTick` immer wieder der Sequenznummer einer dieser Dateneinheiten. Falls die Möglichkeit besteht, daß durch Schalten von `Tick` die Auslieferung der Dateneinheit verhindert wird, ist die Aktion `fNoTick(d)` für alle Daten `d` der Gruppe schaltbar. Deshalb wird `fNoTick` aufgrund seiner starken Fairness irgendwann schalten, wenn immer wieder Daten der Gruppe gesendet werden. Dadurch wird jedoch ein Flag `stdel` gesetzt, das das Schalten der Aktion `Tick` und damit das Inkrementieren von `maxTick` ausschließt. Aufgrund der schwachen Fairness von `fIn` ist dann sichergestellt, daß die Dateneinheit mit der Sequenznummer `maxTick` an den Dienstnehmer ausgeliefert wird.

Das SC *LiveIn* erfüllt nicht die Anforderungen aus Abschnitt 4.3.4 an Livenessprozesse. Das Fehlen der zu `fNoTick` komplementären Aktion `nNoTick` stellt keine Verletzung dieser Anforderungen dar, da die beiden Aktionen aus einer Aktion `NoTick` hergeleitet sind, die die Formel

```
stdel′ = IF (Enabled(fNoTick(d))) THEN "yes" ELSE stdel
```

verwendet. Danach entspricht `nNoTick` aber einem Spontanübergang und kann weggelassen werden, da cTLA$^+$-Spezifikationen invariant gegen Stotterschritte sind. Ein Verstoß ist allerdings die Bedingung `krq` $\leq$ `maxTick` in Aktion `nIn`. Der Grund für die Einführung dieser Safetybedingung liegt darin, daß das SC *LiveIn* sonst nicht mit dem Safety-SC *Reorderings* verträglich wäre. Ohne die Safetybedingung könnte immer eine Dateneinheit mit einer Sequenznummer größer als `maxTick` ausgeliefert werden, so daß das SC *Reorderings* von da an die Auslieferung der Dateneinheit mit der Sequenznummer `maxTick` verbieten würde. Die Schaltbedingung der fairen Aktion `fIn` wäre somit schwächer als die der mit ihr gekoppelten Aktionen, wodurch die in Abschnitt 4.3.4 gestellte Anforderung an die Livenessprozesse verletzt würde. Um die Anforderungen an die Livenessprozesse einhalten zu können, müssen wir

das gewünschte Verhalten durch die Komposition eines Safety-SC und eines Liveness-SC beschreiben. Das Liveness-SC entspricht *LiveIn* ohne die Bedingung `krq` $\leq$ `maxTick` in Aktion `nIn`. Es hält also die Anforderungen aus Abschnitt 4.3.4 ein. Das Safety-SC stimmt ebenfalls mit `LiveIn` überein, wobei allerdings die Fairnessanforderungen weggelassen und die Aktionen `fIn` und `nIn` zu `In` sowie `fNoTick` und das nicht dargestellte `nNoTick` zu `NoTick` zusammengefaßt wurden. Außerdem enthält die Aktion `In` die Schaltbedingung `krq` $\leq$ `maxTick`. Da die beiden SCs nur einen Sinn ergeben, wenn sie miteinander komponiert sind, haben wir sie zu einer Spezifikation zusammengefaßt.

Die SCs *LiveIn* und *LiveInNoAttr* modellieren das gleiche Verhalten, wenn im Ausgabedatenstrom keine Lücken zulässig sind (`tg = 0` $\vee$ `maxrg = 0`). Die Variablen `maxTick` und `maxIn` haben dann immer denselben Wert und `fNoTick` ist niemals schaltbar. *LiveIn* subsumiert also *LiveInNoAttr*, das wir jedoch im Framework als eigenständiges SC beibehalten, da es für einen in der Realität häufig auftretenden Spezialfall eine wesentlich einfachere Spezifikation darstellt.

LiveIn stellt auch die zweite Livenesseigenschaft, daß der Sender immer wieder neue Dateneinheiten senden darf, sicher. Es ist mit der Ausnahme von SC *DataChanOpenR* mit allen SCs kompatibel.

Die dritte Livenessanforderung, daß alle noch unbestätigten Dateneinheiten, deren Wert kleiner als `maxIn` ist, irgendwann implizit bestätigt werden (Aktion `Cf`), wird durch das Liveness-SC *LiveCf* sichergestellt. Dazu besitzt die Aktion `fCf`, die die Bestätigung derartiger Dateneinheiten modelliert, schwache Fairness.

5.2.3 Transfer von Signalisierungsdaten

In Abschnitt 5.1.2 wurde bereits erläutert, daß die SCs des Datentransfers auch zur Modellierung der Transferaspekte im Verbindungsmanagement verwendet werden. Anstatt von Nutzdaten des Typs `usd` wird der Transfer von Signalisierungsdaten des Typs `sigd` beschrieben. `sigd` setzt sich aus verschiedenen Komponenten zusammen (z.B. Dienstbezeichner, Dienstgüteinformation), wird aber, um die Frameworkprozesse flexibel zu halten, nicht als cTLA$^+$-Record spezifiziert. Der Zugriff auf die einzelnen Komponenten wird durch Funktionen modelliert, die in den SCs als Prozeßtypparameter definiert sind und deshalb an reale Datenstrukturen angepaßt werden können. Die Funktionen für `sigd` werden in Abschnitt 5.2.4 erläutert.

Wie einleitend erwähnt wurde, existieren zwei Gruppen an Signalisierungskanaltypbeschreibungen, die sich darin unterscheiden, ob die modellierten Dienste vom Dienstnehmer oder vom Diensterbringer initiiert werden. Kanaltypspezifikationen für vom Dienstnehmer initiierte Dienste entsprechen bis auf den Typ der übertragenen Daten den Datentransferkanalbeschreibungen. Zur Unterscheidung verwenden wir für diese Spezifikationen allerdings die Aktionsbezeichner `SRq`, `SIn` und `SCf`.

Die andere Gruppe betrifft Kanäle zum Management von Verbindungsabbrüchen, die vom Diensterbringer initiiert werden (siehe Abschnitt 2.2.3). Zur Verdeutlichung wird `Rq` in `PAbI` (*Provider-Abort (Initiator)*) und `In` in `PAbP` (*Provider-Abort (Peer)*) umbenannt.

Unterschiede zu den Datentransferkanalspezifikationen bestehen bei der Modellierung von Livenesseigenschaften. In Transferprotokollen werden verlorengegangene oder verfälschte Nutzdaten solange wiederholt gesendet, bis sie fehlerfrei ausgeliefert werden können. Signalisierungsdaten werden hingegen in allen uns bekannten Protokollen nur eine bestimmte Anzahl

oft oder über einen bestimmten Zeitraum hinweg wiederholt. Wenn diese Grenzen überschritten sind, wird die Verbindung abgebrochen. Deshalb werden im Normalfall zur Beschreibung der Signalisierungskanäle keine Liveness-SCs verwendet. Eine Ausnahme bildet hier lediglich der vom Diensterbringer initiierte Verbindungsabbruch, der zwingend beiden Verbindungsteilnehmern angezeigt werden muß. Wir verwenden die Liveness-SCs deshalb nur für die Beschreibung des Kanaltyps für Provider-Aborts.

5.2.4 Safety-SCs der Intraconnection-Coordination

Die im folgenden vorgestellten SCs dienen zur Modellierung des Öffnens und Schließens der Kanäle des Datentransfers und der Übertragung von Signalisierungsdaten in Abhängigkeit von den Diensten der Verbindungsverwaltung. Man kann sie in drei Gruppen unterteilen. Die erste Gruppe betrifft das Öffnen und Schließen der Signalisierungsdatenkanäle und besteht nur aus dem SC *ConnectionState*. Die Verwaltung von Datentransferkanälen wird durch die SCs der zweiten Gruppe beschrieben. Die dritte Gruppe modelliert das vom Dienstnehmer geforderte Verhalten (siehe Abschnitt 5.1.2). Die SCs der Intraconnection-Coordination beeinflussen das Verhalten an genau einem Dienstzugangspunkt.

Zunächst stellen wir den Aufbau der Signalisierungsdaten des Typs `sigd` vor. Mit Hilfe der Funktion `ssigtype : sigd` $\rightarrow$ `SigdTypes` (durch `SigdTypes` werden Kennungen aller Dienstleistungen beschrieben) kann der Diensttyp ermittelt werden, zu dem eine Signalisierungsinformationseinheit gehört.

Wie schon in Abschnitt 2.2.3 erläutert wurde, gibt es unterschiedliche Möglichkeiten, Dienste zu bestätigen. Neben impliziten Diensten ohne Bestätigungen wird meist 2-Way- oder 3-Way-Handshake verwendet[8]. Um Bestätigungsstrategien möglichst flexibel spezifizieren zu können, wurden die Funktionen `sHStot` und `sHSno` eingeführt. `sHStot : sigd` $\rightarrow$ `hskey` (`hskey` steht für eine endliche Teilmenge der natürlichen Zahlen) modelliert die Anzahl an Datenübertragungen, die zur Erbringung eines Dienstes notwendig sind. (Bei impliziten Diensten wird der Wert `1` und bei i-Way-Handshake der Wert i eingesetzt.) Die Funktion `sHSno : sigd` $\rightarrow$ `hskey` gibt an, welche dieser Übertragungen durch die aktuelle Informationseinheit erbracht wird.

Eine wesentliche Aufgabe von Verbindungsverwaltungsdiensten ist das Öffnen oder Schließen von Datentransferkanälen, die durch Kennungen aus der Menge `Qual` identifiziert werden. Durch die Funktionen `sQualWnew` und `sQualRnew` gibt man an, welche Kanäle bei der Sendung oder dem Empfang einer Signalisierungsdateneinheit geöffnet werden sollen. Dabei steht `sQualWnew : sigd` $\rightarrow$ `SUBSET(Qual)` für Datenkanäle, die die gleiche Übertragungsrichtung wie der Signalisierungskanal, über den `sigd` übertragen wird, haben. `sQualRnew : sigd` $\rightarrow$ `SUBSET(Qual)` bezieht sich auf Datenkanäle in umgekehrter Richtung. Zur Schließung von Datenkanälen werden dazu analog die Funktionen `sQualWold` und `sQualRold` verwendet. Beim Öffnen und Schließen von Datenkanälen ist auch von Bedeutung, ob die Datenkanäle schon durch den originalen Dienst oder erst durch eine seiner Bestätigungen geöffnet oder geschlossen werden. Durch `sOpenIW : sigd` $\rightarrow$ `hskey` und `sOpenIR : sigd` $\rightarrow$ `hskey` gibt man an, in welcher Signalisierungseinheit eines Dienstes die Datenkanäle auf der Seite des Dienstinitiators zum Senden (`sOpenIW`) bzw. Ausliefern (`sOpenIR`) geöffnet werden. Der Wert `0` zeigt an, daß durch den Dienst keine neuen Datenkanäle geöffnet werden. Die Öffnung der

[8] Komplexer ist der gesicherte Verbindungsabbau in XTP (siehe oben links in Abbildung 6.1), den man als 4-Way-Handshake auffassen kann.

Sende- bzw. Auslieferungsberechtigung auf der Seite des Partners wird durch die Funktionen `sOpenPW` und `sOpenPR` beschrieben. Zum Schließen von Datenkanälen dienen dazu analog die Funktionen `sCloseIW`, `sCloseIR`, `sClosePW` und `sClosePR`. In Abschnitt 5.2.1 stellten wir das SC *ShowDelivery* vor, durch das modelliert wird, daß der Sender einer Dateneinheit implizit über deren Auslieferung bei seinem Partner unterrichtet wird. Die Funktionen `sOpenmode` und `sClosemode` vom Datentyp `sigd` →`{"immediate", "delayed"}` geben an, ob die Datenkanäle sofort (Wert `'immediate'`) oder erst nach Eintreffen der Auslieferungsbestätigung (Wert `'delayed'`) geöffnet oder geschlossen werden.

Die Funktion `sClosemodeW : sigd → {"immediate", "delayed"}` zeigt an, ob ein Datenkanal sofort nach Eintreffen der Informationseinheit (Wert `'immediate'`) oder erst nach Auslieferung aller gesendeter Dateneinheiten (Wert `'delayed'`) geschlossen wird. Mit der Funktion `sfinal : sigd → hskey` wird angezeigt, welche Informationseinheit eines Dienstes eine Verbindung endgültig beendet. Nach der Übertragung dieser Informationseinheit werden keine weiteren Daten- oder Informationseinheiten in der aktuellen Verbindung mehr gesendet.

Wir stellen nun das SC *ConnectionState* vor, das beschreibt, welche Aktionen der Signalisierungskanäle im aktuellen Zustand der Verbindung geschaltet werden dürfen:

```
PROCESS ConnectionState    Zustand der Verbindung
                        ( sigd : any;   Signalisierungsinformation
                          beginSt : {"Initiator", "Partner", "Indifferent"};
                            Rolle des SAP beim Verbindungsaufbau
                          sHSno : SET[ sigd → hskey ];
                            Nummer der Nachricht im Handshake
                          sfinal : SET[ sigd → hskey ] )
                            Nummer der Verbindungsabbaunachricht
IMPORT Constants;
BODY
  VARIABLES
    SRqSt, SInSt, Other : {"idle", "active", "finished"};
      Status der Aktionen SRq, SIn und aller anderen Aktionen
  INIT ≙ SRqSt = IF (beginSt = "Partner") THEN "idle" ELSE "active" ∧
         SInSt = IF (beginSt = "Initiator") THEN "idle" ELSE "active" ∧
         Other = "idle";
  ACTIONS
    SRq ( s : sigd ) ≙ Uebergabe von Signalisierungsinformation durch den Dienstnehmer
      SRqSt = "active" ∧
      SRqSt′ = IF (sfinal[s] = sHSno[s]) THEN "finished" ELSE "active" ∧
      SInSt′ = IF (sfinal[s] = sHSno[s]) THEN "finished" ELSE "active" ∧
      Other′ = IF (sfinal[s] = sHSno[s]) THEN "finished" ELSE "active";
    SIn ( s : sigd ) ≙ Auslieferung von Signalisierungsinformation an den Dienstnehmer
      SInSt = "active" ∧
      SRqSt′ = IF (sfinal[s] = sHSno[s]) THEN "finished" ELSE "active" ∧
      SInSt′ = IF (sfinal[s] = sHSno[s]) THEN "finished" ELSE "active" ∧
      Other′ = IF (sfinal[s] = sHSno[s]) THEN "finished" ELSE "active";
    SCf ( s : sigd ) ≙
      Implizite Bestaetigung der Auslieferung von Signalisierungsinformation an den Dienstnehmer
      Other = "active" ∧
      SRqSt′ = SRqSt ∧ SInSt′ = SInSt ∧ Other′ = Other;
    PAbI ≙ Verbindungsabbruchnachricht am SAP, an dem die Verbindung abgebrochen wurde
      Other = "active" ∧
```

```
      SRqSt′ = "finished" ∧ SInSt′ = "finished" ∧ Other′ = "finished";
    PAbP ≜ Verbindungsabbruchnachricht am SAP des Partners
      Other = "active" ∧
      SRqSt′ = "finished" ∧ SInSt′ = "finished" ∧ Other′ = "finished";
END
```

Die Zeichenketten `'idle'`, `'active'` und `'finished'` geben an, ob die Verbindung noch nicht aufgebaut wurde, gerade aktiv ist oder schon beendet wurde. Mit Hilfe der Variablen `SRqSt`, `SInSt` und `Other`, die außer im Initialzustand immer denselben Wert haben, wird festgelegt, daß die Aktionen `SRq`, `SIn`, `SCf`, `PAbI` und `PAbP` nur in aktiven Verbindungen geschaltet werden dürfen.

Um einen überlagerten Verbindungsaufbau auszuschließen, der von beiden Dienstzugangspunkten fast gleichzeitig angefordert wird, kann man spezifizieren, daß eine Verbindung nur von einem bestimmten Dienstzugangspunkt initiiert wird. Dazu kann man dem Dienstzugangspunkt, dem das SC zugeordnet ist, durch den Prozeßtypparameter `beginSt` eine bestimmte Rolle zuweisen. Die Rolle `'Initiator'` zeigt an, daß nur dieser Dienstzugangspunkt eine Verbindung aufbauen darf, so daß initial die Aktion `SRq` schaltbar ist, um den Verbindungsaufbauwunsch absetzen zu können. In der Rolle `'Partner'` darf nur der Partner eine Verbindung initiieren, so daß die Aktion `SIn` am Anfang schaltbar ist. Die Rolle `'Indifferent'` erlaubt den Verbindungsaufbau durch beide Dienstzugangspunkte und schließt Überlagerungen nicht aus. In diesem Fall sind `SRq` und `SIn` zu Beginn schaltbar.

Nach der ersten Ausführung von `SRq` oder `SIn` werden alle anderen Aktionen der Verbindungsverwaltung schaltbar. Dieser Zustand bleibt bestehen, bis entweder ein vom Diensterbringer initiierter Abbruch (Provider-Abort) oder ein von einem Dienstnehmer initiiertes Verbindungsende (`sfinal[s] = sHsno[s]`) angezeigt wird und die Aktionen nicht mehr ausgeführt werden dürfen. Wenn in einer Verbindung mehr als zwei Signalisierungskanäle in jeder Übertragungsrichtung spezifiziert werden, müssen die Aktionen `SRq` und `SIn` dupliziert werden (siehe Abschnitt 4.3.3). Das gilt auch für die im folgenden genannten SCs.

Als nächstes skizzieren wir die zweite Gruppe an SCs, durch die die Verwaltung der Datentransferkanäle modelliert wird. Das SC *DataChanOpenW* spezifiziert, ob Nutzdaten über einen Datenkanal gesendet werden dürfen. Für jeden Datenkanal in einer Verbindung wird eine eigene Instanz dieses SCs verwendet, die dem Dienstzugangspunkt auf der Seite des Senders zugeordnet ist. Durch den Prozeßtypparameter `lq` wird der Kanaltyp aus der Menge `Qual` angegeben. Bei der Sendung oder dem Empfang einer Signalisierungsdateneinheit wird die Aktion `Rq` des Datenkanals entweder schaltbar oder nicht schaltbar (Funktionen `sOpenIW`, `sOpenPW`, `sCloseIW` und `sClosePW`). Diese Schaltbedingung, die initial `False` ist, kann im Verlauf der Verbindung beliebig häufig geändert werden. Bei einem Provider-Abort wird die Sendeberechtigung in allen Datenkanälen aufgehoben. Bei einem von einem Dienstnehmer initiierten Verbindungsabbau hat der Dienstnehmer dagegen selbst dafür Sorge zu tragen, daß über keinen Datentransferkanal mehr gesendet werden darf.

Das SC *DataChanOpenR* hat einen gleichartigen Aufbau. Es ist der Empfängerstation eines Datenkanals zugeordnet und modelliert die Zulassung, Daten auszuliefern (Aktion `In`).

Durch das SC *DelayedClose* kann man modellieren, daß die Empfangsberechtigung an einem Kanal durch SC *DataChanOpenR* erst aufgehoben wird, wenn zuvor alle gesendeten Nutzdaten ausgeliefert wurden, wobei am Ende eine Lücke von höchstens `tg`-vielen Dateneinheiten (SC *Gaps* in Abschnitt 5.2.1) auftreten darf.

Das SC *StopOld* dient zur Sicherstellung, daß nach dem erneuten Gewähren der Empfangs-

berechtigung keine Dateneinheiten mehr ausgeliefert werden, die vor der zwischenzeitlichen Schließung gesendet wurden.

Die übrigen Intraconnection-Coordination-SCs gehören zu der dritten Gruppe und erlauben die Festlegung des Verhaltens der Dienstnehmer. Durch das SC *UserBehaviour* kann man festlegen, wann ein Dienstnehmer im Verlauf einer Verbindung neue Dienste initiieren darf. Die Prozeßtypparameter beschreiben einen endlichen Zustandsautomaten (siehe Abschnitt 3.1.3), der das Verhalten des Dienstnehmers modelliert. Durch den Parameter `States` wird die Menge der Zustände des Automaten festgelegt. Der Parameter `InitialStates` beschreibt eine Teilmenge dieser Menge und definiert damit die Anfangszustände. Die erlaubten Zustandsübergänge werden durch die Parameter `TransdSRq`, `TransdSIn`, `TransdSCf`, `TransdRq`, `TransdIn` und `TransdCf` definiert, die sich auf die Aktionen `SRq`, `SIn`, `SCf`, `Rq`, `In` oder `Cf` beziehen. Die Zustandsübergänge von `TransdSRq` und `TransdSIn` hängen im Gegensatz zu den anderen Gruppen nicht nur vom Ausgangszustand, sondern auch von der gesendeten bzw. ausgelieferten Signalisierungsinformation ab.

Durch das SC *UserHSReaction* wird festgelegt, ob und wie der Dienstnehmer eine ankommende Dienstanforderung bestätigen muß. Die Abhängigkeit zwischen empfangenen und in der Bestätigung gesendeten Signalen steht in einigen Fällen fest. Zum Beispiel muß der Wert der Funktion `sHSno` in der Bestätigung um eins höher als der in der empfangenen Nachricht sein. Die Abhängigkeit der anderen Funktionen wird durch den Prozeßtypparameter `HSRule` angegeben. Die Auflösung der Probleme bei einem Verbindungsaufbau, der gleichzeitig von beiden Dienstnehmern initiiert wurde, wird durch den Parameter `SolveContention` beschrieben.

5.2.5 Liveness-SCs der Intraconnection-Coordination

Die Initiierung eines Verbindungsabbruchs, falls eine Signalisierungsdateneinheit auf Dauer nicht ausgeliefert werden kann, ist die einzige Livenesseigenschaft, die in Spezifikationen der Intraconnection-Coordination berücksichtigt werden muß. Sie wird durch das SC *LivePAbI* beschrieben:

```
PROCESS LivePAbI Verbindungsabbruch, wenn Signalisierungsinformation nicht ausgeliefert wird
IMPORT Constants;
BODY
  VARIABLES
    scRq, smaxIn : key; Sequenznummern gesendeter und ausgelieferter Signalisierungsinformation
  INIT ≙ scRq = 0 ∧ smaxIn = 0;
  ACTIONS
    SRq ≙ Uebergabe von Signalisierungsinformation durch den Dienstnehmer
      scRq′ = scRq + 1 ∧ smaxIn′ = smaxIn;
    pSIn ≙ Auslieferung von Signalisierungsinformation an den Dienstnehmer des Partner-SAPs
      smaxIn′ = IF (krq = "notsent")
                  THEN smaxIn ELSE max(smaxIn, krq + 1) ∧
      scRq′ = scRq;
    fPAbI ( krq : key ) ≙ Ausloesung eines Verbindungsabbruchs (fair)
      scRq > smaxIn ∧ krq = smaxIn ∧
      scRq′ = scRq ∧ smaxIn′ = smaxIn;
    nPAbI ( krq : key ) ≙ Ausloesung eines Verbindungsabbruchs (nicht fair)
      not(scRq > smaxIn ∧ krq = smaxIn) ∧
      scRq′ = scRq ∧ smaxIn′ = smaxIn;
```

```
  WF : fPAbI;    Fairnessanforderung
END
```

Das SC referenziert beide Dienstzugangspunkte einer Verbindung, beeinflußt jedoch nur den Dienstzugangspunkt auf der Seite des Senders einer Signalisierungsdateneinheit. Die Aktion `pSIn`[9] ist dem Empfänger der Signalisierungsdaten zugeordnet. Die anderen Aktionen gehören dagegen zum Sender. Wie bei den Daten-SCs wird jeder Informationseinheit eine Sequenznummer zugewiesen, was durch die Variablen `scRq` und `smaxIn` modelliert wird. Durch die schwache Fairness der Aktion `fPAbI` wird ein Verbindungsabbruch garantiert, wenn die zuletzt gesendete Signalisierungsdateneinheit nicht ausgeliefert werden kann. Verbietet man in der Kanalspezifikation Lücken im Ausgabedatenstrom (SC *Gaps* mit `tg = 0`), wird die Verbindung vom Diensterbringer beendet, wenn er irgendeine Signalisierungsinformationseinheit nicht ausliefern kann.

5.2.6 SCs der Interconnection-Coordination

Die zwei SCs *OpenConnections* und *FindDestination* spezifizieren die Aspekte der Koordinierung unterschiedlicher Verbindungen. Das SC *OpenConnections* modelliert, daß an jeder Station eines Rechnernetzes gleichzeitig nur höchstens eine bestimmte Anzahl an Verbindungen aktiv sein darf:

```
PROCESS OpenConnections   Begrenzung der an einem Rechnernetzknoten aktiven Verbindungen
                        ( sigd : any;        Signalisierungsinformation
                          sHSno : SET[ sigd → hskey ];
                            Nummer der Nachricht im Handshake
                          sfinal : SET[ sigd → hskey ];
                            Nummer der Verbindungsabbaunachricht
                          cona, conb : SUBSET( Nat );
                            Menge der Verbindungen, deren SAP A bzw. B zum Knoten gehoeren
                          consize : Nat )   Menge aktiver Verbindungen am Knoten
IMPORT Constants;
BODY
  VARIABLES
    opcon : SUBSET( Nat );   Menge aktiver Verbindungen
  INIT ≙ opcon = ∅;
  ACTIONS
    aSRq ( no : Nat; s : sigd ) ≙
      Uebergabe von Signalisierungsinformation durch den Dienstnehmer an SAP A der Verbindung no
      ( no ∉ cona ∨ no ∈ opcon ∨ card(opcon) < consize ) ∧
      opcon′ = IF (no ∈ cona) THEN
                  IF (sfinal[s] = sHSno[s])
                  THEN opcon \ {no} ELSE opcon ∪ {no}
               ELSE opcon;
    bSRq ( no : Nat; s : sigd ) ≙
      Uebergabe von Signalisierungsinformation durch den Dienstnehmer an SAP B der Verbindung no
      ( no ∉ conb ∨ no ∈ opcon ∨ card(opcon) < consize ) ∧
      opcon′ = IF (no ∈ conb) THEN
```

[9] Wir fügen in den SCs der Intraconnection-Coordination und der Interconnection-Coordination immer den Buchstaben `p` als Präfix bei den Bezeichnern der Aktionen ein, die dem Dienstzugangspunkt zugeordnet sind, der vom SC nicht beeinflußt wird.

```
                IF (sfinal[s] = sHSno[s])
                THEN opcon \ {no} ELSE opcon ∪ {no}
              ELSE opcon;
  aSIn ( no : Nat; s : sigd ) ≙
    Auslieferung von Signalisierungsinformation an den Dienstnehmer an SAP A der Verbindung no
    ( no ∉ cona ∨ no ∈ opcon ∨ card(opcon) < consize ) ∧
    opcon' = IF (no ∈ cona) THEN
                IF (sfinal[s] = sHSno[s])
                THEN opcon \ {no} ELSE opcon ∪ {no}
              ELSE opcon;
  bSIn ( no : Nat; s : sigd ) ≙
    Auslieferung von Signalisierungsinformation an den Dienstnehmer an SAP B der Verbindung no
    ( no ∉ conb ∨ no ∈ opcon ∨ card(opcon) < consize ) ∧
    opcon' = IF (no ∈ conb) THEN
                IF (sfinal[s] = sHSno[s])
                THEN opcon \ {no} ELSE opcon ∪ {no}
              ELSE opcon;
  aPAbI ( no : Nat ) ≙
    Verbindungsabbruchnachricht an SAP A, ausgeloest an A
    opcon' = IF (no ∈ cona) THEN opcon \ {no} ELSE opcon;
  bPAbI ( no : Nat ) ≙
    Verbindungsabbruchnachricht an SAP B, ausgeloest an B
    opcon' = IF (no ∈ conb) THEN opcon \ {no} ELSE opcon;
  aPAbP ( no : Nat ) ≙
    Verbindungsabbruchnachricht an SAP A, ausgeloest an B
    opcon' = IF (no ∈ cona) THEN opcon \ {no} ELSE opcon;
  bPAbP ( no : Nat ) ≙
    Verbindungsabbruchnachricht an SAP B, ausgeloest an A
    opcon' = IF (no ∈ conb) THEN opcon \ {no} ELSE opcon;
END
```

Die Beschreibung des gesamten Transportsystems wurde durch ein Prozeßfeld der Verbindungsbeschreibungen erzeugt. Wie in Abschnitt 4.3.2 gezeigt wurde, wird durch diese Operation jeder Aktion ein zusätzlicher Parameter `no` hinzugefügt, durch den man die einzelnen Prozesse des Prozeßfeldes referenziert. `no` dient somit als eindeutige Verbindungskennung. Da jede Verbindungsbeschreibung zwei Dienstzugangspunkte referenziert, modelliert das Prozeßfeld zwei Gruppen von Dienstzugangspunkten. Durch die Buchstaben `a` und `b` vor den Aktionsnamen wird angegeben, welchem Dienstzugangspunkt einer Verbindung die Aktion zugeordnet ist. Die Struktur des Rechnernetzes wird durch das SC *OpenConnections* spezifiziert. Für jede Station *St* des Rechnernetzes wird eine Instanz des SCs gebildet. Der Prozeßtypparameter `cona` gibt die Menge aller Verbindungsbezeichner `no` an, deren Dienstzugangspunkte der Gruppe *a* dem Knoten *St* zugeordnet sind. Analog dazu bezeichnet `conb` die zur Station *St* gehörenden Dienstzugangspunkte der Gruppe *b*. Die Aktionen der Gruppe *a* werden mit den Aktionen des SCs *OpenConnections* gekoppelt, deren Bezeichner den Präfix `a` enthalten. Die Aktionen mit dem Präfix `b` werden mit den Aktionen des Prozeßfelds, die der Gruppe *b* zugeordnet sind, gekoppelt. Die Funktion `consize` gibt an, wieviele Verbindungen maximal am Knoten *St* gleichzeitig aktiv sein können. Die momentan offenen Verbindungen werden in der Mengenvariablen `opcon` gespeichert. Eine neue Verbindung darf nur geöffnet werden, wenn die Kardinalität von `opcon` nicht den Wert `consize` übersteigt. Um die Bildung einer

SC-Instanz für jeden Knoten des Rechnernetzes zu vereinfachen, kann man zunächst ein Prozeßfeld über *OpenConnections* bilden, das man anschließend mit der Verbindungsspezifikation komponiert.

Die Verbindungsauswahl wird durch das SC *FindDestination* modelliert. Es stellt sicher, daß beim Neuaufbau immer eine Verbindung gewählt wird, deren Dienstzugangspunkte zum einen an der eigenen Station und zum anderen am gewünschten Zielknoten angeordnet sind. Der Prozeßtypparameter `sadr` beschreibt die Menge aller Stationsadressen. Die Kennungen der Verbindungen an der eigenen Station wird durch den Prozeßtypparameter `con` (analog zu `cona` und `conb` in SC *OpenConnections*) modelliert. Der Prozeßtypparameter `adrlist` spezifiziert die Abhängigkeit zwischen den Adressen der anderen Stationen und den Verbindungskennungen. Die Empfangsadresse wird vom Dienstnehmer als Teil der Signalisierungsinformation aus der Menge `sigd` (siehe Abschnitt 5.2.4) angegeben, wobei der Zugriff auf die Empfängeradresse durch die Funktion `sdest` beschrieben wird.

5.3 Abstract Protocol Mechanisms (APMs)

Die abstrakten Protokollmechanismen (APMs) modellieren im Gegensatz zu den SCs die räumliche Verteilung der Systemressourcen. Sie verwenden im Gegensatz zu detaillierteren FAPMs vereinfachte und meist nicht implementierbare Kontrollstrukturen wie eindeutige Datensequenznummern und unendlich große Datenpuffer. Ein Protokoll besteht aus zwei Protokollinstanzen, die miteinander über einen untergeordneten Dienst kommunizieren. Wie bei den SCs betrachten wir nur Protokolle, die Datentransfer zwischen zwei Kommunikationspartnern erbringen.

Die Protokollinstanzen setzen sich aus Protokollmechanismen zusammen. Durch Interaktion von Protokollmechanismen aus mehreren Instanzen werden Protokollfunktionen realisiert, die bestimmte Diensteigenschaften erfüllen. Die Struktur der Protokollmechanismen wurde bereits in Abschnitt 5.1.3 erläutert (für APMs des Datentransfers siehe auch Abbildung 5.3). Jede Spezifikation eines Protokollmechanismus (APM, Abstract Protocol Mechanism) ist genau einem Dienstzugangspunkt zugeordnet. Die Komposition aller APMs an einem Dienstzugangspunkt bildet die Beschreibung einer Protokollinstanz. Die APMs der beiden Instanzen S (Sender) und R (Empfänger) interagieren über die Basiskanäle M (Daten von S nach R) und A (Daten von R nach S), die durch die AMCs (Abstract Media Constraints; siehe Abschnitt 5.4) modelliert werden. Die Interaktion mit den Dienstnehmern wird durch die Aktionen `Rq` (Senden von Daten), `In` (Ausliefern von Daten) und `Cf` (implizites Bestätigen ausgelieferter Daten) spezifiziert. `Rq` und `Cf` sind der Sendeinstanz S und `In` der Empfangsinstanz R zugeordnet. Der Zugriff auf die Basiskanäle erfolgt durch die Aktionen `MRq` und `ARq` (Senden) sowie `MIn` und `AIn` (Empfangen).

Die Instanzen übertragen über die Basiskanäle Protokolldateneinheiten (PDUs) aus der Menge `pdu`. `pdu` ist in den APMs als Prozeßtypparameter definiert und kann deshalb an reale PDU-Formate angepaßt werden. Der Zugriff auf die Komponenten der PDU wird wie beim Datentyp `sigd` durch Funktionen modelliert, die ebenfalls als Prozeßtypparameter definiert sind. PDUs bestehen aus der Protokollsteuerinformation (PCI) und eventuell aus Nutzdaten. Die PCI wird durch einen Datentyp `pci` beschrieben. In den APMs modelliert die Funktion `spci : pdu` $\rightarrow$ `pci` den Zugriff auf die PCI einer PDU. Die Referenzierung der Nutzdaten

wird durch die Funktion[10] `susd : (pdu × pkey) → usd` beschrieben. Die einzelnen Komponenten der PCI werden ebenfalls durch Funktionen spezifiziert, die wir im Zusammenhang mit den APM-Beschreibungen einführen werden.

In den Abschnitten 5.3.1 und 5.3.2 stellen wir zunächst die Safety- und Liveness-APMs für den Datentransfer vor. Die APMs für die Intraconnection-Coordination werden in Abschnitt 5.3.3 und diejenigen der Interconnection-Coordination in Abschnitt 5.3.4 erläutert.

5.3.1 Safety-APMs des Datentransfers

Die APMs des Datentransfers modellieren abstrakte Protokollmechanismen. Sie erbringen die in Abschnitt 2.2.2 beschriebenen Protokollfunktionen unter Verwendung von Kontrollstrukturen, die unendlich viele Zustände annehmen dürfen. Nicht alle der im folgenden dargestellten APMs spezifizieren Protokollmechanismen, die für die Realisierung der durch die Safety-SCs (siehe Abschnitt 5.2.1) modellierten Diensteigenschaften zwingend notwendig sind. Einige Protokollmechanismen (z.B. die Flußkontrolle) dienen nämlich nicht der Erbringung spezieller Diensteigenschaften, sondern schließen die Übertragung von für den Protokollfortschritt nicht notwendigen Daten aus und erhöhen damit die effektive Datenrate. Die APMs, die diese Mechanismen spezifizieren, sind nicht zur Verifikation von Safety-SCs jedoch zur Verifikation von Liveness-SCs notwendig, da sie die Schaltbedingungen fairer Aktionen einschränken. Um Livenessbeweise im Framework zu vereinfachen, wurden diese APMs dem Framework hinzugefügt und in den Theoremen zum Beweis von Liveness-SCs berücksichtigt. Andere durch instanzlokale Mechanismen realisierbare Protokollfunktionen wie Übertragungswiederholungen, die von einem Zeitgeber im Sender abhängen, oder Flußkontrolle durch temporäre Sendebeschränkungen schränken die Liveness des Protokolls dagegen nicht ein, da sie auf Zeitgebern beruhen, die die Übertragung von PDUs nach einem endlichen Zeitraum nicht mehr blockieren. Sie stellen nur Realzeitanforderungen dar, die in dieser Arbeit noch nicht berücksichtigt werden. Für diese Funktionen werden deshalb keine APMs angeboten.

Im folgenden stellen wir zunächst die APMs vor, durch die die Safety-SCs des Datentransfers (siehe Abschnitt 5.2.1) implementiert werden. Dabei handelt es sich zum einen um APMs, die fehlerfreie Übertragung und Auslieferung von Dateneinheiten modellieren. Zum anderen werden APMs eingeführt, die Bestätigungsmechanismen für beim Empfänger eingetroffene Nutzdaten beschreiben und damit die Verwendung endlicher Übertragungskapazitäten erlauben.

Wir erläutern zunächst die APMs *SBufferKey* und *RBufferKey*, die die Verwaltung der Sequenznummern übertragener Dateneinheiten modellieren. Sie werden in allen Protokollspezifikationen eingesetzt und dienen zur Implementierung des SCs *SDUId* sowie zahlreicher anderer SCs:

```
PROCESS SBufferKey    Sequenznummernverwaltung an der Sendeinstanz S
                    ( pdu, pci, usd : Any CONSTANT ;  PDUs, PCIs und Nutzdaten
                      encpdu : SUBSET(pdu) ;          Menge unverfaelschter PDUs
                      spci : SET[pdu → pci] ;         Zugriff auf die PCI einer PDU
                      skey : SET[pci → (SUBSET(pkey) ∪ {"≪≫"})] ;
                        Zugriff auf die Sequenznummern uebertragener Nutzdaten in einer PCI
                      skk : SET[pkey → key] ;
                        Sequenznummer der Dateneinheit, zu der das Segment gehoert
```

[10] Der Bezeichner `pkey` verweist auf Nutzdatenkennungen, die später vorgestellt werden.

```
                    skn : SET[pkey → Nat] ; Interne Segmentnummer
                    skm : SET[pkey → Nat] ;
                      Anzahl an Segmenten einer Dateneinheit
                    usdsize : SET[usd → Nat] ;
                      Groesse von Dateneinheiten (wieviele Segmente sind notwendig?)
                    mb : Nat ∪ {"infty"} )   Maximale Anzahl an Segmenten in einer PDU
IMPORT Constants;
BODY
VARIABLES
    cRq : key;            Sequenznummernzaehler
    sbk : SUBSET(pkey); Menge verwendeter Segmentkennungen
 INIT ≙ cRq = 0 ∧ sbk = ∅ ;
 ACTIONS
    Rq (krq : key; d : usd) ≙ Uebergabe vom Dienstnehmer
      krq = cRq ∧
      sbk′ = sbk ∪ { k | k ∈ pkey : skk[k] = krq ∧
                                    skn[k] < usdsize[d] ∧
                                    skm[k] = usdsize[d] } ∧
      cRq′ = cRq + 1;
    MRq ( krq : (SUBSET(pkey) ∪ {"≪≫"}); p : pdu ) ≙
      Sendung ueber Basiskanal M
      p ∈ encpdu ∧ krq = skey[spci[p] ] ∧
      ( krq = "≪≫" ∨
        ( krq ⊆ sbk ∧
          (card(krq) ≤ mb ∨ mb = "infty") ) ) ∧
      cRq′ = cRq ∧ sbk′ = sbk;
END
```

Das APM *SBufferKey* ist der Instanz S zugeordnet und modelliert die Sequenznummerverwaltung durch den Sender, die aus der Vergabe neuer Sequenznummern, der Segmentierung von Nutzdaten, dem Ausschluß der Übermittlung nicht gesendeter Nutzdaten, der Blockung von Nutzdaten und der Redundanzsicherung der gesendeten Protokolldateneinheiten (PDUs) besteht. Die Vergabe der Sequenznummern erfolgt wie in SC *SDUId* mit Hilfe des Zählers `cRq`. Jeder neu gesendeten Nutzdateneinheit wird in aufsteigender Reihenfolge eine Sequenznummer `krq` aus der Menge `key` zugewiesen, die sich aus den natürlichen Zahlen und dem Zusatzelement `'notsent'` (Phantome) zusammensetzt[11].

Die Segmentierung von Nutzdaten in kleinere Einheiten, die voneinander separat übertragen und beim Empfänger wieder zu den originalen Nutzdaten zusammengesetzt werden[12], wird durch die Prozeßtypparameter `skk`, `skn`, `skm` und `usdsize` beschrieben. Die Datensegmente werden durch Kennungen aus der Menge `pkey` identifiziert. `skk[k]` modelliert eine Referenz zwischen einer Segmentkennung `k` und der Sequenznummer der Nutzdateneinheit, zu der das Segment `k` gehört. Alle Segmente einer Nutzdateneinheit erhalten eindeutige Indizes, die durch die Funktion `skn[k]` angegeben werden, so daß durch `skk` und `skn` eine eindeutige Segmentkennung bestimmt wird. Für die Reassemblierung der Segmente in der Empfängerinstanz R muß diese die Anzahl an Segmenten kennen, aus denen eine Nutzdaten-

[11] Man kann leicht beweisen, daß die Variablen der APMs, die der Protokollinstanz S zugeordnet sind, niemals den Wert `' n` annehmen. Zur annehmen. Zur besseren Übersicht verwenden wir dennoch in diesen APMs die Menge `key` anstatt `Nat`.

[12] Für die Übertragung größerer Nutzdaten über zellenbasierte Übertragungsverfahren wie ATM, das pro Zelle nur 48 Nutzdatenbytes erlaubt (siehe Abschnitt 2.4.1), ist Datensegmentierung erforderlich.

einheit besteht. Diese Information ist durch die Funktion `skm` aus der Kennung von Segmenten ablesbar. Der Prozeßtyp `usdsize[d]` legt fest, in wieviele Segmente eine Dateneinheit zerlegt werden muß. Bei der Segmentierung (Aktion `Rq`) werden die für Übertragung der Nutzdateneinheit `d` mit der Sequenznummer `krq` notwendigen Segmentkennungen erzeugt und in den Sendespeicher `sbk` eingefügt.

Die Aktion `MRq` modelliert die Sendung einer PDU `p` über den Basiskanal M zum Empfänger. Da Blockung unterstützt wird, können in einer PDU mehrere Datensegmente übermittelt werden. Die Menge der Kennungen der in PDU `p` übertragenen Segmente wird durch den Aktionsparameter `krq` angezeigt und ist in der Protocol Control Information (PCI) der PDU durch die Funktion `skey` abrufbar. In PDUs, die nicht der Datenübertragung dienen, besitzt `skey` den Wert `'<<>>'`. Die Aktion `MRq` stellt sicher, daß nur Segmente einer vom Dienstnehmer bereits gesendeten Dateneinheit in der PDU enthalten sein dürfen (`krq` $\subseteq$ `sbk`). Außerdem ist festgelegt, daß ein Block höchstens `mb`-viele Dateneinheiten umfaßt, wenn der Prozeßtypparameter `mb` nicht mit dem Wert `'infty'` instantiiert ist.

Die Redundanzsicherung durch ein „Cyclic Redundancy Code"-Verfahren (vgl. [LKK93, Tan89]) zur Überprüfung von übertragenen PDUs auf Bitfehler wird durch die Menge `encpdu` abstrahiert, die als Prozeßtypparameter definiert ist. `encpdu` bezeichnet die Menge aller PDUs, die eine korrekte Prüfsumme besitzen. Um dem Empfänger zu ermöglichen, anhand der Prüfsumme die Korrektheit einer PDU festzustellen, muß der Sender für jede übertragene PDU eine korrekte Prüfsumme generieren. Nach diesem Modell dürfen deshalb nur PDUs aus `encpdu` gesendet werden.

Das APM *RBufferKey* liegt auf der Empfängerseite R und modelliert die Überprüfung empfangener PDUs auf Bitfehler, die Zwischenspeicherung empfangener Segmente unter Berücksichtigung der Empfangspuffergröße, die Reassemblierung der Segmente zu Nutzdaten und die Auslieferung von Daten an den Dienstnehmer:

```
PROCESS RBufferKey   Sequenznummernverwaltung an der Empfangsinstanz R
                   ( pdu, pci, usd : Any CONSTANT ;  PDUs, PCIs und Nutzdaten
                     encpdu : SUBSET(pdu) ;          Menge unverfaelschter PDUs
                     spci : SET[pdu → pci] ;         Zugriff auf die PCI einer PDU
                     skey : SET[ pci → (SUBSET(pkey) ∪ {"≪ ≫"})] ;
                       Zugriff auf die Sequenznummern uebertragener Nutzdaten in einer PCI
                     skk : SET[pkey → key] ;
                       Sequenznummer der Dateneinheit, zu der das Segment gehoert
                     skn : SET[pkey → Nat] ;         Interne Segmentnummer
                     skm : SET[pkey → Nat] ;         Anzahl an Segmenten einer Dateneinheit
                     usdsize : SET[usd → Nat] ;
                       Groesse von Dateneinheiten (wieviele Segmente sind notwendig?)
                     rcu : Nat ∪ {"infty"} ;   Groesse des
                     rcc : Nat )               Empfangspuffer
IMPORT Constants;
BODY
  VARIABLES
    rbk : SUBSET(pkey) ;  Menge empfangener Segmentkennungen
    maxIn : key ;         Hoechste Sequenznummer der ausgelieferten Daten
  INIT ≙ rbk = ∅ ∧ maxIn = 0;
  ACTIONS
    In ( krq : key; d : usd ) ≙   Auslieferung an den Dienstnehmer
      ( { k | k ∈ pkey : skk[k] = krq ∧ skn[k] < usdsize[d] ∧
```

```
                          skm[k] = usdsize[d] } ⊆ rbk  ∨
      ( krq = "notsent"  ∧  krq ∈ rbk ) ) ∧
    maxIn′ = IF (krq = "notsent") THEN maxIn ELSE max(maxIn, krq + 1) ∧
    rbk′ = rbk;
  MIn ( krq : (SUBSET(pkey) ∪ {"≪≫"}); p : pdu ) ≙   Empfang von Basiskanal M
    krq = IF (p ∈ encpdu)
          THEN skey[spci[p ] ] ELSE {"notsent"} ∧
    rbk′ = IF (krq ≠ "≪≫" ∧
               (krq = {"notsent"} ∨
                rcu = "infty" ∨
                card( { skk[k] | k ∈ pkey : k ∈ (rbk ∪ krq) } ∩
                          { k  | k ∈ key : k > maxIn + rcc}) ≤ rcu) )
          THEN rbk ∪ krq
          ELSE rbk ∧
    maxIn′ = maxIn ;
END
```

Jede empfangene PDU (Aktion `MIn`) wird durch die Überprüfung, ob sie sich in der Menge `encpdu` befindet, auf Bitfehler untersucht. Falls eine PDU `p` korrekt ist, werden alle in `skey[ spci[p] ]` enthaltenen Segmentkennungen im Empfangsspeicher `rbk` zwischengespeichert, wenn dadurch nicht dessen Kapazität überschritten wird. Die Kapazität, die durch die Prozeßtypparameter `rcu` und `rcc` festgelegt wird, berücksichtigt nur Segmente von Nutzdaten, deren Sequenznummer größer oder gleich dem Wert der Variablen `maxIn` ist. Wie im SC *SDUId* beschreibt `maxIn` die Sequenznummer der Dateneinheit, die bei Einhaltung der Ausgabereihenfolge dem Dienstnehmer als nächstes ausgeliefert werden muß. `rbk` darf nur Segmentkennungen von höchstens `rcu`-vielen Nutzdateneinheiten mit Kennungen größer als `maxIn + rcc` enthalten[13]. Durch `rcu = "infty'` werden Empfangsspeicher von unbegrenzter Kapazität angezeigt. Der Aktionsparameter `krq` der Aktion `MIn` gibt die Kennungen der in der PDU `p` enthaltenen Segmente an. Fehlerhafte PDUs werden als Phantome aufgefaßt, für die der Wert `'notsent'` in den Empfangspuffer `rbk` eingetragen wird.

Eine Nutzdateneinheit `d` mit der Sequenznummer `krq` darf nur an den Dienstnehmer ausgeliefert werden (Aktion `In`), wenn sie ein Phantom kennzeichnet (`krq = "notsent'`) oder wenn die Kennungen aller ihrer Segmente im Empfangspuffer `rbk` gespeichert sind, was durch das Feld `skm` in den Segmentkennungen festgestellt wird.

Im Gegensatz zu den beiden oben vorgestellten APMs, die nur die Verwaltung der Segmentkennungen modellieren, wird die Verwaltung der Nutzdaten und der Datensegmente durch die APMs *SBufferUsd* und *RBufferUsd* spezifiziert. In diesen APMs beschreiben die Variablen `sbu : pkey → usd` und `rbu : pkey → usd` den Zugriff auf die Speicher zu sendender bzw. empfangener Datensegmente. Die eigentliche Segmentierung und Reassemblierung der Daten wird durch den Prozeßtypparameter `usdsplit : (Nat × usd) → usd` modelliert. (`usdsplit[(i,d)]` beschreibt den Inhalt des `i`-ten Segments der Dateneinheit `d`.) Während der Übertragung sind die Segmente im Nutzdatenbereich einer PDU enthalten, der durch die Funktion `susd` referenziert wird. Die APMs *SBufferUsd* und *RBufferUsd* implementieren in Komposition mit *SBufferKey* und *RBufferKey* die SCs *Corruptions* bzw. *RelativeCorruptions*.

[13] Die Beschreibung der Empfangspuffergröße durch zwei Parameter `rcc` und `rcu` erlaubt darzustellen, daß für einen zusammenhängenden Bereich an Sequenznummern oberhalb von `maxIn` Speicherplatz zur Verfügung steht. Dadurch werden Verklemmungen aufgrund eines überfüllten Empfangsspeichers ausgeschlossen.

Wie schon in Abschnitt 5.2.1 festgestellt wurde, beeinflussen die durch die SCs *Duplicates*, *Reorderings*, *Gaps* und *Phantoms* modellierten Diensteigenschaften nur das Verhalten des Dienstzugangspunktes des Empfängers R und können somit durch lokale Protokollmechanismen in R realisiert werden. Das Framework besitzt APMs *RDuplicates*, *RReorderings*, *RGaps* und *RPhantoms*, die dasselbe Verhalten wie die (mit Ausnahme des Präfix R) gleichnamigen SCs beschreiben.

Das APM *RBufferKey* stellt nicht sicher, daß Lücken im Ausgabedatenstrom nur Dateneinheiten betreffen, die fehlerhaft übertragen wurden. Wir führen deshalb das APM *RVoluntaryGaps* ein, das ein Überspringen korrekt und vollständig empfangener Dateneinheiten bei Auslieferungen an den Dienstnehmer ausschließt. Es wird zur Implementierung des SC *RelativeGaps* verwendet.

Die Begrenzung der Sendespeicherkapazität und damit der Gesamtkapazität von Protokollen wird durch drei Protokollmechanismen realisiert, die durch die APMs *RAcknowledge*, *SAcknowledge* und *SCapacity* spezifiziert werden. *RAcknowledge* und *SAcknowledge* modellieren die Bestätigung der korrekt empfangenen Datensegmente, während durch *SCapacity* die Anzahl gesendeter, aber noch nicht bestätigter Dateneinheiten im Sendepuffer `sbk` beschränkt wird.

Die Übertragung der Kennungen von empfangenen Segmenten in PDUs von Instanz R nach Instanz S über den Basiskanal A wird durch das APM *RAcknowledge* spezifiziert. Wir unterstellen, daß nach der Auslieferung einer Nutzdateneinheit mit der Kennung `krq` die spätere Auslieferung von Nutzdateneinheiten mit kleineren Kennungen als `krq` nicht mehr garantiert werden kann (siehe auch Liveness-SC *LiveIn* in Abschnitt 5.2.2). Deshalb ist es sinnvoll, diese Segmente zu bestätigen, um den Sendepuffer nicht durch nicht mehr notwendige Datensegmente zu blockieren. In jeder Bestätigungs-PDU sind also alle Segmentkennungen von Daten mit Sequenznummern enthalten, die kleiner als `maxIn` sind. Daneben kann auch eine Anzahl an Kennungen von empfangenen, aber noch nicht ausgelieferten Nutzdaten bestätigt werden. Die maximale Anzahl noch nicht ausgelieferter Nutzdaten wird durch den Prozeßtypparameter `ma` festgelegt. (Bei `ma = "infty'` existiert keine derartige Begrenzung.) Die Funktion `sack : pci` $\rightarrow$ `SUBSET(pkey)` gibt die Referenz auf den Bereich der PCI einer PDU an, in der die bestätigten Kennungen enthalten sind. In PDUs ohne Bestätigungen hat `sack` den Wert `'<<>>'`.

Im APM *SAcknowledge* werden durch Instanz S bestätigte Segmentkennungen in einem Puffer `sba` abgespeichert (Aktion `AIn`). Bestätigte Datensegmente dürfen nicht mehr zur Instanz R gesendet werden.

Die Begrenzung der Sendepuffergröße in Instanz S wird durch das APM *SCapacity* modelliert. Die vom Partner bestätigten Segmentkennungen werden wie im APM *SAcknowledge* in Puffer `sba` abgelegt. Eine neue Nutzdateneinheit darf vom Dienstnehmer nur gesendet werden (Aktion `Rq`), wenn dadurch die Anzahl unbestätigter Dateneinheiten eine durch den Prozeßtypparameter `sc` angegebene Schranke nicht überschreitet.

Das APM *SShowDelivery* modelliert, daß der Sender nur dann eine Bestätigung für eine Dateneinheit erhält (Aktion `Cf`), wenn deren Segmente vollständig bestätigt wurden. Außerdem darf jede Dateneinheit nur einmal bestätigt werden. Das APM implementiert in Komposition mit *RAcknowledge* und *SAcknowledge* das SC *ShowDelivery*.

Mit den oben vorgestellten APMs kann man alle Safety-SCs des Datentransfers (siehe Abschnitt 5.2.1) implementieren. Wir führen im folgenden weitere APMs ein, die Protokollmechanismen zur Einschränkung der Sendung nicht notwendiger Daten spezifizieren. Dazu

gehören Kreditvergabeschemata, explizite Zurückweisungen fehlerhafter Daten, explizite Anforderungen von Protokollsteuerinformation durch die Partnerinstanz und die Ordnung des Stroms gesendeter Daten.

Wir stellen zunächst zwei APMs für Funktionen der Flußkontrolle durch Kreditvergabe mit absoluten Fenstern vor (siehe Abschnitt 2.2.2), die auch die Rücknahme zuvor gegebener Kredite erlauben (Reneging; vgl. [DDK+90]). Das APM *RCredit* modelliert die Sendung einer Sequenznummer `k` aus der Menge `key` durch die Empfängerinstanz R, die damit die Sendung der Segmente aller Nutzdaten freigibt, deren Sequenznummer nicht größer als `k` ist. Durch die Funktion `scre` wird die Position einer Kreditmitteilung innerhalb einer PCI beschrieben. Die Höhe des Kredits muß mindestens der Kapazität des Empfangspuffers (siehe APM *RBufferKey*) entsprechen, der alle Dateneinheiten bis zu der Sequenznummer `maxIn + rcu + rcc` speichern kann. Um Laufzeitverzögerungen zu berücksichtigen, kann der Empfänger jedoch auch für zusätzliche Dateneinheiten Kredit geben, die er allerdings bei verzögerter Auslieferung von Daten an den Dienstnehmer nicht speichern kann und deshalb verwerfen muß. Die maximale Anzahl nicht speicherbarer Dateneinheiten wird durch den Prozeßtypparameter `mo` limitiert, so daß die Kreditobergrenze nicht größer als `maxIn + rcu + rcc + mo` sein darf.

Ein zuvor bereits gegebener Kredit, der in der Spezifikation durch Variable `rcred` beschrieben ist, darf nur innerhalb der durch den Prozeßtypparameter `mr` definierten Grenze zurückgenommen werden (Reneging). Die erlaubte Kredituntergrenze beträgt somit `rcred - mr`. Beliebiges Reneging wird durch `mr = "infty'` spezifiziert. Allerdings darf die Kredituntergrenze niemals den Wert `maxIn + rcc` unterschreiten, da sonst Verklemmungen entstehen können.

Die Verwaltung von Sendekrediten beim Sender S wird durch das APM *SCredit* modelliert. Beim Empfang einer PDU vom Partner wird der Sendekredit im Zähler `scred` gespeichert. Daten-PDUs dürfen nur Segmente von Nutzdaten enthalten, deren Sequenznummern kleiner oder gleich dem Wert von `scred` sind. Initial wird der Kredit auf den Wert des Prozeßtypparameters `mcr` gesetzt.

Protokollmechanismen zur Übertragungswiederholung durch selektive Zurückweisung werden durch die APMs *RReject* und *SReject* modelliert. Das APM *RReject* beschreibt, daß die Empfängerinstanz R die Wiederholung beliebiger Segmente verlangen kann, die nicht korrekt empfangen wurden. Sie sendet dazu in der PCI einer über Medium A (siehe Abschnitt 5.1.3) übertragenen PDU eine Menge an Segmentkennungen aus `pkey`, die durch die Funktion `snak : pci` $\rightarrow$ `SUBSET(pkey)` referenziert wird. In *SReject* wird modelliert, daß der Sender S nur Datensegmente übertragen darf, die er entweder noch niemals gesendet hat oder die von R explizit zur Wiederholung angefordert wurden.

Die explizite Anforderung von Empfängerreaktionen wie die Sendung von Bestätigungen, Wiederholungsanforderungen und Kreditvergabe wird durch die APMs *STriggerAck*, *STriggerCredit*, *RTriggerAck* und *RTriggerCredit* modelliert. Die APMs *STriggerAck* und *STriggerCredit* spezifizieren die Versendung dieser Anforderungen durch S. Dazu werden in der PCI einer über Kanal M gesendeten PDU die Flags `stack` bzw. `stcre` auf einen der Werte `'on'` oder `'off'` gesetzt. Das APM *RTriggerAck* modelliert, daß die Versendung von Bestätigungen bzw. Wiederholungsanforderungen vom Empfänger R nur ausgeführt wird, wenn er zuvor eine PDU mit auf `'on'` gesetztem Flag `stack` erhalten hat. *RTriggerCredit* beschreibt, daß der Empfänger eine neue Kreditmitteilung nur sendet, wenn in einer empfangenen PDU das Flag `stcre` auf `'on'` gesetzt war.

Das APM *SSendInOrder* stellt sicher, daß ein Segment einer Nutzdateneinheit `k` nur

zum Partner gesendet wird (Aktion `MRq`), wenn zuvor schon Segmente von allen Nutzdaten gesendet wurden, deren Sequenznummern kleiner als `k` sind. Die erstmalige Sendung von Nutzdaten über den Kanal M erfolgt somit in der durch den Dienstnehmer vorgegebenen Reihenfolge.

5.3.2 Liveness-APMs des Datentransfers

Wir stellen im folgenden die vier Liveness-APMs *SLiveMRq*, *RLiveARq*, *RLiveIn* und *SLiveCf* vor, die die in Abschnitt 5.2.2 eingeführten Liveness-SCs implementieren. Diese APMs modellieren Protokollmechanismen, die die geforderte Liveness des Transportdienstes unter Verwendung von Basiskanälen mit wesentlich geringeren Livenessanforderungen erbringen. In der Realität garantiert ein Basisdienst überhaupt keine Livenessanforderungen, da er unter Umständen auf Dauer gestört sein kann. Ohne eine Livenessanforderung kann man jedoch nicht die Liveness des Zieldienstes beweisen. Wir nehmen deshalb an, daß die Basiskanäle M und A die folgende sehr schwache Livenessanforderung erfüllen: Jede Informationseinheit, die immer wieder über einen Basiskanal gesendet wird, wird ihrem Empfänger irgendwann ausgeliefert. Ein derartiges Verhalten kann man durch die AMCs *MLiveIn* und *ALiveIn* beschreiben, die dem in Abschnitt 5.2.2 vorgestellten SC *LiveIn* entsprechen. Bei der Modellierung der Liveness-Protokollmechanismen muß man sicherstellen, daß die Sendeinstanz S alle Datensegmente und die Empfangsinstanz R alle Bestätigungen bzw. Wiederholungsanforderungen solange wiederholt, bis sie garantiert ausgeliefert wurden.

Bei der Entwicklung von Livenessbeschreibungen, die zu allen Safety-APMs kompatibel sind, halten wir die in Abschnitt 4.3.4 beschriebenen Prinzipien ein. Wir modellieren möglichst minimale Livenessanforderungen. Bei einem Transportprotokoll ist die Auslieferung aller Daten bereits sichergestellt, wenn es sich so passiv wie ein Stop-and-Wait-Protokoll verhält, in dem eine neue Dateneinheit durch die Sendeinstanz S erst gesendet wird, wenn S eine Bestätigung ihres Vorgängers erhalten hat.

Die Liveness-APMs werden so ausgelegt, daß sich ein Transportprotokoll nicht passiver als ein Stop-and-Wait-Protokoll verhält. Der Sender S muß immer wieder ein beliebiges noch nicht bestätigtes Segment der ältesten nicht vollständig bestätigten Nutzdateneinheit mit der Sequenznummer i übertragen. Durch die Livenessanforderung an den Basiskanal M wird sichergestellt, daß das Segment irgendwann beim Empfänger R eintrifft.

R muß eine Nutzdateneinheit nur ausliefern, wenn ihre Sequenznummer der Nachfolger der höchsten Sequenznummer aller bisher ausgelieferten Dateneinheiten ist. Wenn die letzte dem Dienstnehmer ausgelieferte Dateneinheit die Sequenznummer j hat, müssen alle Segmente der Nutzdateneinheiten mit Sequenznummern zwischen $j+1$ und $i-1$ im Empfangspuffer von R sein, da sie nach Definition von i beim Sender bereits bestätigt wurden. Die Fairness der Auslieferung stellt sicher, daß irgendwann das Nutzdatum mit der Kennung $i-1$ an den Dienstnehmer ausgeliefert wird, so daß danach die Dateneinheit i zur Auslieferung vorgesehen ist.

R muß die Segmente aller Nutzdaten bestätigen, deren Sequenznummern kleiner als die der letzten ausgelieferten Dateneinheit j sind. Außerdem müssen an S Bestätigungen für alle korrekt empfangenen Segmente von j übermittelt werden. Nach Auslieferung von $i-1$ werden also die Kennungen der korrekt empfangenen Segmente von Dateneinheit i bestätigt. Nach wiederholtem Senden der Bestätigungen werden diese aufgrund der Liveness des Basiskanals A irgendwann beim Sender S eintreffen, der von da ab ein anderes Segment von i sendet. Auf

diese Weise treffen irgendwann alle Segmente von Nutzdatum i beim Empfänger R ein, der i dann an den Dienstnehmer ausliefert. Nach Erhalt der Bestätigung aller Segmente sendet S die Segmente des Nutzdatums mit der Kennung $i + 1$. Somit werden nacheinander alle Nutzdaten an den Empfänger übertragen und dem Dienstnehmer ausgeliefert, wenn beim Empfänger keine Lücken im Ausgabedatenstrom auftreten dürfen.

In Abschnitt 4.3.4 wird gefordert, daß die Liveness-APMs zu allen in Abschnitt 5.3.1 skizzierten Safety-APMs kompatibel sind. Die Schaltbedingungen der fairen Aktionen dürfen nicht schwächer sein als die Schaltbedingungen der mit ihnen gekoppelten Aktionen aus den Safetyprozessen. Die Liveness-APMs wurden so konstruiert, daß sie mit Ausnahme des APMs *DataChanOpenR* (siehe Abschnitt 5.2.2) mit allen Safety-APMs komponiert werden können[14]. Die Möglichkeit, daß in Liveness-APMs unter Umständen PDU-Felder verwendet werden, die in den ausgewählten Safety-APMs nicht definiert sind, wirft keine Probleme auf, da diese PDU-Felder durch konstante Werte oder durch Kombinationen anderer PDU-Felder modelliert werden können (siehe Beispiel XTP in Abschnitt 6.2.2). Ist in einer Protokollspezifikation zum Beispiel kein Sendekreditvergabemechanismus definiert, kann man das in den APMs *SLiveMRq* und *RLiveARq* verwendete PDU-Feld `scre` durch den kleinsten Wert aller Sequenznummern von Nutzdaten ersetzen, deren Segmente nicht vollständig in PDU-Feld `sack` enthalten sind.

Zunächst stellen wir das APM *SLiveMRq* vor, das der Sendeinstanz S zugeordnet ist und die Übertragung von Datensegmenten zur Empfangsinstanz R spezifiziert:

```
PROCESS SLiveMRq    Liveness der Sendung von Daten ueber Basiskanal M
                  ( pdu, pci, usd : Any CONSTANT ;  PDUs, PCIs und Nutzdaten
                    encpdu : SUBSET(pdu) ;          Menge unverfaelschter PDUs
                    spci : SET[pdu → pci] ;         Zugriff auf die PCI einer PDU
                    skey : SET[pci → (SUBSET(pkey) ∪ {"≪ ≫"})] ;
                        Zugriff auf die Sequenznummern uebertragener Nutzdaten in einer PCI
                    sack : SET[ pci → (SUBSET( pkey) ∪ {"≪ ≫"})] ;
                        Zugriff auf die Sequenznummern bestaetigter Nutzdaten in einer PCI
                    snak : SET[pci → (SUBSET(pkey) ∪ {"≪ ≫"})] ;
                        Zugriff auf die Sequenznummern zurueckgewiesener Nutzdaten in einer PCI
                    scre : SET[ pci → (key ∪ {"≪ ≫"})] ;
                        Zugriff auf die Kreditanzeige in einer PCI
                    stack : SET[ pci → {"on","off"}] ;
                        Zugriff auf die Bestaetigungswunschanzeige in einer PCI
                    stcre : SET[ pci → {"on","off"}] ;
                        Zugriff auf die Kreditwunschanzeige in einer PCI
                    skk : SET[pkey → key] ;
                        Sequenznummer der Dateneinheit, zu der das Segment gehoert
                    skn : SET[pkey → Nat] ;         Interne Segmentnummer
                    skm : SET[pkey → Nat] ;         Anzahl an Segmenten einer Dateneinheit
                    usdsize : SET[usd → Nat] ;
                        Groesse von Dateneinheiten (wieviele Segmente sind notwendig ?)
                    usdsplit : SET[ (Nat × usd) → usd] ; Segmentierungsfunktion
                    mcr : Nat ;                     Kredit am Anfang
                    rcc : Nat )                     Groesse des Empfangspuffers
IMPORT Constants;
```

[14] Dabei müssen allerdings die Prozeßtypparameter der Safetyprozesse durch dieselben Werte ersetzt werden wie diejenigen der Fairnessprozesse.

```
BODY
VARIABLES
    cRq : key;                  Sequenznummernzaehler
    sbk : SUBSET(pkey);         Menge verwendeter Segmentkennungen
    sba : SUBSET(pkey);         Menge von Kennungen bestaetigter Segmente
    sbu : SET[pkey → usd] ;     Zu Segmentkennungen gehoerige Nutzdaten
    sbr : SUBSET(pkey);         Menge von Kennungen zurueckgewiesener Segmente
    scred : key;                Kreditobergrenze
 INIT ≙ cRq = 0 ∧ sbk = ∅ ∧ sba = ∅ ∧ sbr = ∅ ∧ scred = mcr ∧
        sbu ∈ SET[pkey → usd] ;
 ACTIONS
   Rq (krq : key; d : usd) ≙ Uebergabe vom Dienstnehmer
     krq = cRq ∧
     sbk′ = sbk ∪ { k | k ∈ pkey : skk[k] = krq ∧
                                   skn[k] < usdsize[d] ∧
                                   skm[k] = usdsize[d] } ∧
     sbu′ = [FCN k ∈ pkey ↦ IF ( skk[k] = krq ∧ skn[k] < usdsize[d] )
                             THEN usdsplit[(skn[k],d)]
                             ELSE sbu[k] ] ∧
     cRq′ = cRq + 1 ∧
     sba′ = sba ∧ sbr′ = sbr ∧ scred′ = scred;
   fMRq ( krq : (SUBSET(pkey) ∪ {"≪ ≫"}); p : pdu;
          kd : {"dt","tc","ta"} ) ≙ Sendung ueber Basiskanal M (fair)
     LET SMinNA: SUBSET(key) ≙
           { k | k ∈ key : ∃ l ∈ pkey :: l ∈ (sbk \ sba) ∧ skk[l] = k };
         klive ≙ IF SMinNA = ∅ THEN cRq ELSE Min(SMinNA);
     IN
       p ∈ encpdu ∧ krq = skey[spci[p] ] ∧
       ( krq = "≪ ≫" ∨
         ( krq ⊆ sbk \ (sba ∪ sbr) ∧
           krq ⊆ { k | k ∈ pkey : skk[k] ≤ scred } ∧
           ∀ k ∈ krq :: susd[(p,k)] = sbu[k] ) ) ∧
       stcre[spci[p] ] ∈ { "on", "off" } ∧
       stack[spci[p] ] ∈ { "on", "off" } ∧
       ( kd = "dt" ⇒
           krq ≠ "≪ ≫" ∧ krq ≠ ∅ ∧
           ∃ k ∈ krq :: skk[k] = klive ∧
           krq ⊆ { k | k ∈ pkey : skk[k] ≤ klive + rcc } ) ∧
       ( kd = "tc" ⇒ ( scred < klive ∧ stcre[spci[p] ] = "on" ) ) ∧
       ( kd = "ta" ⇒ ( klive < cRq ∧ stack[spci[p] ] = "on" ) ) ∧
       sbr′ = IF (krq = "≪ ≫") THEN sbr ELSE (sbr ∪ krq) ∧
       cRq′ = cRq ∧ sbk′ = sbk ∧ sba′ = sba ∧ scred′ = scred ∧
       sbu′ = sbu;
   nMRq ( krq : (SUBSET(pkey) ∪ {"≪ ≫"}); p : pdu;
          kd : {"dt","tc","ta"} ) ≙ Sendung ueber Basiskanal M (nicht fair)
     LET SMinNA: SUBSET(key) ≙
           { k | k ∈ key : ∃ l ∈ pkey :: l ∈ (sbk \ sba) ∧ skk[l] = k };
         klive ≙ IF SMinNA = ∅ THEN cRq ELSE Min(SMinNA);
     IN
     not(
       p ∈ encpdu ∧ krq = skey[spci[p] ] ∧
```

```
        ( krq = "≪ ≫" ∨
          ( krq ⊆ sbk \ (sba ∪ sbr) ∧
            krq ⊆ { k | k ∈ pkey : skk[k] ≤ scred } ∧
            ∀ k ∈ krq :: susd[(p,k)] = sbu[k] ) ) ∧
        stcre[spci[p] ] ∈ { "on", "off" } ∧
        stack[spci[p] ] ∈ { "on", "off" } ∧
        ( kd = "dt" ⇒
            krq ≠ "≪ ≫" ∧ krq ≠ ∅ ∧
            ∃ k ∈ krq :: skk[k] = klive ∧
            krq ⊆ { k | k ∈ pkey : skk[k] ≤ klive + rcc } ) ∧
        ( kd = "tc" ⇒ ( scred < klive ∧ stcre[spci[p] ] = "on" ) ) ∧
        ( kd = "ta" ⇒ ( klive < cRq ∧ stack[spci[p] ] = "on" ) ) ) ∧
      sbr′ = IF (krq = "≪ ≫") THEN sbr ELSE (sbr ∪ krq) ∧
      cRq′ = cRq ∧ sbk′ = sbk ∧ sba′ = sba ∧ scred′ = scred ∧
      sbu′ = sbu;
  AIn ( p : pdu ) ≙ Empfang von Basiskanal A
      sba′ = IF (p ∈ encpdu ∧ sack[spci[p] ] ≠ "≪ ≫")
             THEN sba ∪ sack[spci[p] ]
             ELSE sba ∧
      sbr′ = IF ( p ∈ encpdu ∧ snak[spci[p] ] ≠ "≪ ≫" )
             THEN sbr \ snak[spci[p] ] ELSE sbr ∧
      scred′ = IF ( p ∈ encpdu ∧ scre[spci[p] ] ≠ "≪ ≫")
               THEN scre[ spci[p] ] ELSE scred ∧
      cRq′ = cRq ∧ sbk′ = sbk ∧ sbu′ = sbu;
  SF: fMRq;  Fairnessanforderung
END
```

Zum einen wird durch das APM sichergestellt, daß der Sender S genügend häufig Segmente der ältesten noch nicht bestätigten Dateneinheit überträgt. Zum anderen sichert das APM zu, daß S genügend häufig Bestätigungen, Wiederholungsanforderungen oder Gewährung eines neuen Sendekredits von der Instanz R anfordert. Da die Schaltbedingung der stark fairen Aktion `fMRq` nicht schwächer als die Schaltbedingung der Aktion `MRq` einer der Safety-APMs sein darf, muß die Funktionalität fast aller Safety-APMs von Instanz S berücksichtigt werden, weshalb das APM *SLiveMRq* recht komplex ist. Aufgrund der in Abschnitt 4.3.4 vorgestellten Konventionen wurde eine Aktion `nMRq` definiert, die eine zu `fMRq` komplementäre Schaltbedingung besitzt. Wir müssen sicherstellen, daß durch `fMRq` gesendete PDUs `p` sowohl genügend viele Datensegmente der ältesten unbestätigten Dateneinheit als auch genügend häufige Anforderungen an R enthalten. Deshalb führen wir neben `krq` und `p` einen weiteren Aktionsparameter `kd` ein, der die Werte `'dt'`, `'tc'` und `'ta'` annehmen darf. Nach der Definition von Prozeßtypen im kompositionalen Spezifikationsstil (siehe Abschnitt 4.1.1) wird für jede Instanz dieser Aktion mit einer der drei Parameterbelegungen von `kd` starke Fairness gefordert. Die Schaltbedingung von `fMRq` stellt für `kd = "dt'` sicher, daß in der gesendeten PDU `p` ein Nutzdatensegment der ältesten unbestätigten Dateneinheit (Bezeichner `klive`) enthalten sein muß. Die Anforderung eines neuen Sendekredits (`kd = "tc'`) wird nur garantiert, wenn für die älteste unbestätigte Dateneinheit kein Kredit mehr vorhanden ist. Bestätigungen müssen immer angefordert werden (`kd = "ta'`), wenn noch nicht alle Segmente der vom Dienstnehmer gesendeten Dateneinheiten bestätigt sind.

Die Garantie, daß der Empfänger R von Dateneinheiten seinem Partner Bestätigungen, Wiederholungsanforderungen und neue Sendekedite übermittelt, wird durch das APM *RLive-*

ARq spezifiziert, dessen Aufbau den gleichen Prinzipien wie derjenige des APMs *SLiveMRq* folgt. Die stark faire Aktion `fARq` besitzt einen zusätzlichen Aktionsparameter `kd`, durch den Fairness für die Sendung von Sendekreditmitteilungen (`kd = "cr'`), für Bestätigungen von Segmenten (`kd = "ak'`) und für Wiederholungsanforderungen (`kd = "nk'`) sichergestellt ist. Wie bereits ausgeführt wurde, reicht es für die Liveness des Protokolls aus, nur die Segmente aller Nutzdaten mit Sequenznummern kleiner als `maxIn` und die empfangenen Segmente von `maxIn` zu bestätigen. Wiederholungsanforderungen müssen nur für alle noch nicht empfangenen Segmente von `maxIn` gesendet werden. Sendekreditmitteilungen müssen nur die untere erlaubte Kreditgrenze (siehe APM *RCredit* im vorhergehenden Abschnitt) enthalten. Das APM *RLiveARq* ist kompatibel zu allen Safety-APMs der Empfängerinstanz R.

Das APM *RLiveIn*, das der Instanz R zugeordnet ist, stellt sicher, daß eine Nutzdateneinheit mit einer Sequenznummer zwischen `maxIn` und `maxIn + tg` irgendwann ausgeliefert wird, wenn alle Segmente dieser Dateneinheit eingetroffen und im Empfangspuffer `rbk` gespeichert sind. Dazu besitzt die Aktion `fIn` schwache Fairness. Da durch alle Safety-APMs die Auslieferung der Nutzdateneinheiten mit Sequenznummern zwischen `maxIn` und `maxIn + tg` garantiert wird, sind sie zum APM *RLiveIn* kompatibel.

Die schwache Fairness der Aktion `fCf` im APM *SLiveCf* garantiert, daß für jede vollständig bestätigte Nutzdateneinheit eine Bestätigungsmeldung an den Dienstnehmer gesendet wird.

5.3.3 APMs der Intraconnection-Coordination

Die Verbindungsverwaltung der Protokolle wird auf die gleiche Art spezifiziert wie die der Dienste (siehe Abschnitt 5.2.3). Die Aspekte der Übertragung von Signalisierungsdaten werden durch die APMs modelliert, die wir in den beiden vorangehenden Abschnitten beschrieben haben. Dabei werden allerdings der Typ `usd` durch den Typ `sigd` der Signalisierungsdaten und die Funktion `susd`, durch die die Nutzdaten in PDUs referenziert wird, durch die Funktion `ssigd : pdu` $\rightarrow$ `sigd` ersetzt. Außerdem fügen wir bei den Aktions- und Variablenbezeichnern ein `s` als Anfangsbuchstaben hinzu. Mit Hilfe dieser APMs kann man Signalisierungsdatenkanäle definieren. Die Koordinierung dieser Kanäle und der Datenkanäle einer Verbindung wird durch weitere APMs modelliert, die die in den Abschnitten 5.2.4 und 5.2.5 vorgestellten SCs der Intraconnection-Coordination realisieren.

Die SCs *ConnectionState*, *DataChanOpenW*, *DataChanOpenR*, *StopOld*, *UserBehaviour* und *UserHSReaction* beeinflussen das Verbindungsverhalten nur an jeweils einem einzigen Dienstzugangspunkt. Wir können aus diesen SCs somit gleichartige APMs ableiten.

Das SC *DelayedClose* modelliert, daß ein Datenkanal erst geschlossen wird, wenn alle zuvor gesendeten Dateneinheiten bereits ausgeliefert wurden[15]. Die Empfangsinstanz R eines Kommunikationsprotokolls kann diese Aufgabe nur erfüllen, wenn sie die Sequenznummer der zuletzt vom Dienstnehmer gesendeten Dateneinheit kennt. Die Übertragung dieser Sequenznummer in der Signalisierungsinformation und die Verzögerung der Kanalschließung werden durch die APMs *SMarkFinal*, *RMarkFinal* und *RDeliverBuff* beschrieben. *SMarkFinal* spezifiziert, daß in der Instanz S das Inkrement der Sequenznummer der zuletzt gesendeten Nutzdateneinheit der Signalisierungsdateneinheit hinzugefügt wird, die das Schließen eines Datenkanals anzeigt. Die Referenz auf diese Information wird durch die Funktion `sclosecRq : (sigd` $\times$ `qual)` $\rightarrow$ `key` beschrieben, wobei `qual` den zu schließenden Datenkanal identifiziert (siehe auch Abschnitt 5.2.4). Durch *RMarkFinal* wird modelliert, daß die

[15] Dabei werden allerdings Lücken im Ausgabestrom toleriert.

Signalisierungsdateneinheit, die den Datenkanal schließt, dem Dienstnehmer erst übergeben wird, wenn (mit Ausnahme tolerierter Lücken) alle Nutzdaten ausgeliefert worden sind. Das APM *RDeliverBuff* spezifiziert, daß ein Datenkanal nur geschlossen werden darf, wenn in seinem Empfangsspeicher keine Dateneinheiten enthalten sind, deren Auslieferung an den Dienstnehmer zulässig ist. Dadurch wird ausgeschlossen, daß die Signalisierungsinformation zuvor eingetroffene Dateneinheiten „überholt".

Das in Abschnitt 5.2.5 vorgestellte Liveness-SC *LivePAbI* garantiert, daß eine Verbindung vom Diensterbringer abgebrochen wird, wenn eine Signalisierungsdateneinheit nicht dem Empfänger ausgeliefert werden kann. Die Feststellung, daß eine Dateneinheit nicht auslieferbar ist, erfolgt am Dienstzugangspunkt des Empfängers, während der Verbindungsabbruch auf der Seite des Senders initiiert wird. Der Protokollmechanismus, der diese Diensteigenschaft realisiert, bricht eine Verbindung ab, wenn für eine gesendete Signalisierungsdateneinheit auf Dauer keine Bestätigung durch die Partnerinstanz R eintrifft. Er wird durch das APM *SLivePAbI* modelliert. Diese Fairnessanforderung ist stärker als die des SCs *LivePAbI*, da im Gegensatz dazu ein Verbindungsabbruch auch in dem Fall erzwungen wird, daß ein Signalisierungsdatum ausgeliefert, aber nicht bestätigt werden kann. Aufgrund der räumlichen Verteilung kann man jedoch keine schwächeren Fairnessanforderungen spezifizieren. Der Bestätigungsmechanismus wird wie beim Datentransfer durch die APMs *SAcknowledge* und *RAcknowledge* (siehe Abschnitt 5.3.1) modelliert.

5.3.4 APMs der Interconnection-Coordination

Die in Abschnitt 5.2.6 beschriebenen SCs haben nur auf das Verhalten eines einzigen Dienstzugangspunktes Einfluß und werden deshalb durch gleichartige APMs *OpenConnections* und *FindDestination* implementiert.

5.4 Abstract Media Constraints (AMCs)

Das Verhalten der Basiskanäle M und A wird durch die Abstrakten Medieneigenschaften (Abstract Media Constraints, AMCs) beschrieben. Da die Basiskanäle Dienste anbieten, können wir zur Modellierung die in Abschnitt 5.2 eingeführten SCs verwenden. Die AMCs werden aus den SCs gebildet, indem den Prozeß-, Aktions-, Parameter- und Variablenbezeichnern die Buchstaben `M` bzw. `A` vorangestellt werden. Als Nutzdaten werden anstatt Elementen vom Typ `usd` Protokolldateneinheiten (PDUs) vom Typ `pdu` übertragen.

Da der durch ein Protokoll erbrachte Zieldienst umfassender als der vom Protokoll verwendete Basisdienst ist, wird der Basisdienst meist durch weniger SCs oder durch eine größere Toleranz gegen Übertragungsfehler modelliert. Die von Transportprotokollen verwendeten meist unzuverlässigen Basisdienste werden in der Regel durch die SCs *SDUId*, *Corruptions*, *Reorderings*, *Capacity*, *Phantoms* und *LiveIn* und zum Teil durch *RelativeGaps*, *RelativeCorruptions* und *RelativePhantoms* beschrieben. Durch *Reorderings* und *Capacity* wird dabei die endliche Lebensdauer[16] der über einen Basiskanal übertragenen PDUs spezifiziert. Die begrenzte Lebensdauer ist insbesondere zur Herleitung der FAPMs (siehe Abschnitt 5.5.1) wichtig. Durch *Corruptions* und *Phantoms* wird modelliert, daß alle ausgelieferten verfälschten

[16]Die Lebensdauer hängt dabei von der Anzahl gesendeter PDUs ab. Zum Beispiel schließt die Sendung einer PDU mit der Sequenznummer `k` die Auslieferung aller PDUs mit Sequenznummern kleiner als `k - c - tr` aus. Damit kann die Lebensdauer ohne Einführung eines Realzeitkonzeptes modelliert werden.

PDUs und Phantome eine falsche Prüfsumme besitzen, anhand deren sie von der Protokollinstanz erkannt werden können. Anderenfalls könnte ein Protokollfehler auftreten. *RelativeGaps*, *RelativeCorruptions* und *RelativePhantoms* werden eingesetzt, wenn das Protokoll keine Fehlerbehebung durch Datenwiederholung verwendet. *LiveIn* sichert die Liveness des Basisdienstes. Mit Hilfe dieser AMCs kann man nachweisen, daß die relative Häufigkeit bestimmter Übertragungsfehler im Zieldienst nicht höher als diejenige in den Basiskanälen ist.

5.5 Finite Abstract Protocol Mechanisms (FAPMs)

Mit den FAPMs werden reale Protokollmechanismen spezifiziert. Da sie nur Variablen mit endlichen Wertebereichen besitzen, können sie relativ einfach in reale Programme umgesetzt werden. Man kann alle Protokollmechanismen, die durch die in Abschnitt 5.3 vorgestellten APMs modelliert werden, auch durch FAPMs beschreiben. Die Einschränkung auf Variablen mit endlichen Wertebereichen betrifft zum einen die Sequenznummerverwaltung in den FAPMs des Datentransfers und der Intraconnection-Coordination, so daß beliebig viele Dateneinheiten mit nur einer endlichen Anzahl an Sequenznummern übertragen werden. Da eine Sequenznummer im Verlauf einer Verbindung unter Umständen mehreren Dateneinheiten zugewiesen wird, muß sichergestellt sein, daß es nicht zu Protokollfehlern durch Verwechslung von Daten mit derselben Sequenznummer kommen kann. Zum anderen müssen die FAPMs der Interconnection-Coordination sicherstellen, daß in einem Rechnernetz beliebig viele Verbindungen aufgebaut werden können, obwohl zu ihrer Identifizierung nur endlich viele Verbindungskennungen zur Verfügung stehen. Auch hier dürfen keine Verwechslungen, bei denen die PDUs einer Verbindung als Phantome in einer anderen auftreten, entstehen.

Im folgenden zeigen wir, daß man das durch die APMs spezifizierte Verhalten auch durch FAPMs beschreiben kann, wenn die Basiskanäle einige Bedingungen erfüllen. Im Abschnitt 5.5.1 bilden wir die FAPMs des Datentransfers und der Intraconnection-Coordination, indem wir zeigen, wie man die Variablen mit unendlichen Wertebereichen der APMs durch Variablen mit endlichen Wertebereichen ersetzen kann, ohne das durch die Prozesse beschriebene Verhalten zu verändern. Auf die Bildung der FAPMs der Interconnection-Coordination wird in Abschnitt 5.5.2 eingegangen.

5.5.1 Entwurf der FAPMs des Datentransfers und der Intraconnection-Coordination

Die APMs des Datentransfers und der Intraconnection-Coordination werden nach dem folgenden Schema in FAPMs überführt: Zuerst zeigen wir, daß man für alle Protokolle, die auf Schiebefenstermechanismen basieren, eine endliche obere Schranke für die Anzahl an Sequenznummern aktiver Dateneinheiten angeben kann[17]. Eine Dateneinheit ist aktiv, wenn sie das Verhalten eines Protokollmechanismus beeinflussen kann. In einem zweiten Schritt skizzieren wir, wie man zu den APMs sich äquivalent verhaltende APMs bilden kann, in denen

[17]Dabei betrachten wir nur allgemeine Schiebefensterprotokolle. Für zwei Spezialfälle, in denen der Basisdienst Vertauschungsfreiheit garantiert bzw. die Empfangsinstanz nur Bestätigungsmeldungen für bereits ausgelieferte Dateneinheiten erzeugt, kann man kleinere obere Schranken angeben. Diese Fälle werden hier nicht genauer erklärt. Das Framework enthält allerdings Theoreme zum Beweis von APMs, die diese Spezialfälle berücksichtigen.

mit der Ausnahme von Historienvariablen (siehe Abschnitt 3.3.4) nur Variablen mit endlichen Wertebereichen enthalten sind. Durch Weglassung der Historienvariablen werden die APMs schließlich in FAPMs überführt.

Wir betrachten im folgenden immer ein System *Sys*, das mindestens aus jeweils einer Instanz der APMs *SBufferKey*, *RBufferKey*, *SAcknowledge*, *RAcknowledge*, *SCapacity* und *SSendInOrder* besteht. Dabei muß die Kapazität des Empfangspuffers endlich sein (`rcu` $\neq$ `"infty'` in *RBufferKey*). Ferner darf die Anzahl an Datensegmenten in einer PDU eine endliche Grenze `mb` nicht überschreiten. Der Übertragungskanal M hat ein endliches Gedächtnis. Er darf Dateneinheiten nur ausliefern, wenn seit ihrer Sendung weniger als `c + tr`-viele neue Dateneinheiten gesendet wurden (AMCs *MReorderings* und *MCapacity*; siehe Abschnitt 5.4). Im Prinzip könnte man auf diese Art auch das endliche Gedächtnis von Kanal A modellieren. Über A werden jedoch keine Nutzdaten sondern nur Datenbestätigungen gesendet. Da Sammelbestätigungen zulässig sind, könnten bei einer niedrigen Bestätigungsrate noch sehr alte Sequenznummern bestätigt werden, so daß eine unrealistisch große Menge aktiver Sequenznummern vorhanden wäre. Wir führen aus diesem Grund ein neues AMC *AReorderingsMRq* ein, durch das das Gedächtnis nicht von den auf Kanal A sondern von den auf Kanal M gesendeten Dateneinheiten abhängt. Das AMC modelliert, daß eine PDU nur ausgeliefert werden darf (Aktion `AIn`), wenn seit ihrer Sendung (Aktion `ARq`) höchstens `tr`-viele PDUs über das Medium M (Aktion `MRq`) gesendet wurden. Dieser Mechanismus orientiert sich an der in [AL91b] vorgeschlagenen Realzeitbeschreibungsmethode, da man eine Sendung durch `MRq` als Fortschreiten einer virtuellen Uhr um eine Zeiteinheit auffassen kann.

Außerdem muß der Datenbestätigungsmechanismus verändert werden. In jeder Bestätigungsmitteilung sind in den APMs *SAcknowledge* und *RAcknowledge* immer die Segmente aller Nutzdaten enthalten, die kleiner als der Wert der Variablen `maxIn` sind. Dadurch sind im Feld `sack` auch Sequenznummern inaktiver Dateneinheiten enthalten. Um in PDUs jedoch nur aktive Sequenznummern speichern zu müssen, ändern wir die Zugriffsfunktion `sack` zur Bestätigung von Segmentkennungen so ab, daß veraltete Sequenznummern nicht mehr separat aufgeführt werden. Der Empfänger gibt in den PDUs eine Sequenznummer `sackm` an und bestätigt damit alle Dateneinheiten, deren Sequenznummern kleiner oder gleich `sackm` sind. Außerdem können Sequenznummern, die größer als `sackm` sind, einzeln bestätigt werden (Funktion `sacko`). Diese Zugriffsfunktion ist in den speziellen APMs *SAcknowledgeFA* und *RAcknowledgeFA* modelliert.

Wir zeigen zunächst, daß man in den APMs der Sendeinstanz S je eine endliche Konstante m_s und M_s angeben kann, so daß alle Sequenznummern aktiver Daten im Intervall [`cRq`$-m_s$, `cRq`$+M_s$] liegen. `cRq` ist eine im APM *SBufferKey* definierte Variable, die den inkrementierten Wert der zuletzt vergebenen Sequenznummer anzeigt. Dazu zeigen wir zunächst in Satz 5.1, daß eine Konstante $\widehat{m_s}$ existiert, so daß alle Dateneinheiten, deren Sequenznummern kleiner als `cRq` $- \widehat{m_s}$ sind, vollständig bestätigt sind. $\widehat{m_s}$ ist allerdings keine untere Schranke der aktiven Daten, da in Bestätigungsmeldungen über Kanal A kleinere Sequenznummern als `cRq` $- \widehat{m_s}$ enthalten sein können. Unter Verwendung von Satz 5.1 weisen wir in Satz 5.2 die Existenz der unteren Schranke m_s für aktive Dateneinheiten nach. In Satz 5.3 verifizieren wir, daß man eine obere Schranke M_s für die Sequenznummern aktiver Dateneinheiten angeben kann.

In Satz 5.1 beweisen wir zunächst, daß alle Nutzdaten mit Sequenznummern, die kleiner als `cRq - sc - rcc - rcu` sind, in Instanz S vollständig bestätigt sein müssen. (`sc` gibt die Größe des Sendepuffers und `rcc + rcu` diejenige des Empfangspuffers an.)

Satz 5.1 Sys *sei aus je einer Instanz der APMs* SBufferKey, RBufferKey, SCapacity, SAcknowledgeFA *und* RAcknowledgeFA *sowie der AMCs* MSDUId, ASDUId, MCorruptions *und* ACorruptions *komponiert. Der Prozeßtypparameter* `ma` *von APM* RAcknowledgeFA *habe die Wertbelegung* `rcc` + `rcu`.

Dann ist

$$\widehat{m_s} \triangleq sc + rcc + rcu$$

eine untere Schranke für unbestätigte Dateneinheiten im Sendepuffer `sbk` *der Instanz* S.

Beweis: Wir bilden zunächst eine Invariante, in der wir die folgenden Abkürzungen verwenden:

- $minU$ ist die kleinste Sequenznummer aller Nutzdaten, deren Bestätigungen noch nicht vollständig beim Sender S eingetroffen sind.
- $sbak$ ist die Menge der Sequenznummern aller in S vollständig bestätigten Nutzdaten. (Alle Segmentkennungen sind in der Variablen `sba` enthalten.)
- AB_i beschreibt die Menge aller Sequenznummern, deren Segmente in der i-ten durch Instanz S von Kanal A empfangenen PDU p bestätigt werden (Funktionen `sacko` und `sackm`).
- ABn_i beschreibt die kleinste Sequenznummer einer Dateneinheit, von der keine Segmentkennung in AB_i enthalten ist.
- cs entspricht der Menge $\{0, \ldots, cARq - 1\}$

Die Invariante lautet[18]

$$\begin{aligned}
& card(\{\texttt{skk}[k] | k \in \texttt{rbk}\}) \leq rcc + rcu \\
\wedge \quad & \forall i \in cs : AB_i \subseteq \{\texttt{skk}[k] | k \in \texttt{rbk}\} \cup \{k | k < \texttt{maxIn}\} \\
\wedge \quad & \forall s \in 2^{cs} : card(\bigcup_{i \in s} AB_i) \leq max_{i \in s}(ABn_i) + rcc + rcu \\
\wedge \quad & \exists s \in 2^{cs} : sbak = \bigcup_{i \in s} AB_i \wedge minU = max_{i \in s}(ABn_i) \\
\wedge \quad & cRq \leq card(sbak) + sc
\end{aligned}$$

Ihr Beweis erfolgt nach dem in Abschnitt 3.3.3 gezeigten Verfahren und bleibt dem Leser überlassen.

Mit Hilfe dieser Invarianten kann man die Behauptung zeigen. Aus ihrem dritten und vierten Konjunkt folgt

$$card(sbak) \leq minU + rcc + rcu \tag{5.1}$$

Aufgrund des fünften Konjunkts gilt damit jedoch auch die Bedingung

$$\texttt{cRq} \leq minU + sc + rcc + rcu \tag{5.2}$$

und damit die Behauptung. □

Mit Hilfe dieses Satzes beweisen wir nun, daß es eine untere Schranke m_s für die aktiven Sequenznummern in der Sendeinstanz S, gibt:

[18] 2^M steht für die Potenzmenge von M.

Satz 5.2 Sys *sei aus je einer Instanz der APMs* SBufferKey, RBufferKey, SCapacity, SAcknowledgeFA, RAcknowledgeFA *und* SSendInOrder *und der AMCs* MSDUId, ASDUId, MCorruptions, ACorruptions, MReorderings, MCapacity *und* AReorderingsMRq *komponiert.*

Dann ist

$$m_s \triangleq 2 \cdot \mathtt{sc} + 2 \cdot \mathtt{rcu} + 2 \cdot \mathtt{rcc} + \mathtt{mb} \cdot \mathtt{tr} + 1$$

eine untere Schranke für in Bestätigungsmeldungen enthaltene Sequenznummern und wegen Satz 5.1 eine untere Schranke aller aktiven Dateneinheiten in der Instanz S.

Beweis: Der Beweis wird aufgrund seiner Komplexität nur skizziert. Er setzt sich aus einer größeren Anzahl einfacher Invariantenbeweise zusammen. Im Verlauf des Beweises müssen Annahmen über einen Systemzustand getroffen werden, der bereits in der Vergangenheit liegt[19]. Wir bezeichnen mit Z_h den Systemzustand, in dem sich *Sys* unmittelbar vor der Sendung der PDU mit der PDU-Kennung $\mathtt{cMRq} - \mathtt{tr}$ über Kanal M befand. Mit Z bezeichnen wir den aktuellen Zustand, den wir zunächst betrachten wollen.

Aufgrund der endlichen Kapazität des Sendepuffers `sbk` (APM *Capacity*) wurde eine Dateneinheit mit der Sequenznummer l, für die

$$l \geq \mathtt{cRq} - \mathtt{sc} - 1 \tag{5.3}$$

gilt, bereits vollständig bestätigt. Da l (oder eine Dateneinheit mit höherer Sequenznummer) dann aber von R empfangen wurde, hat S zuvor mindestens ein Segment dieser Dateneinheit über den Kanal M gesendet (Aktion `MRq`).

Das AMC *AReorderingsMRq* modelliert, daß nur PDUs an S ausgeliefert werden (Aktion `AIn`), wenn nach ihrer Sendung (Aktion `ARq`) die Aktion `MRq` höchstens `tr`-mal geschaltet wurde. Somit können also nur Bestätigungen ausgeliefert werden, die nicht vor dem Systemzustand Z_h durch R gesendet wurden.

Wir betrachten nun den Systemzustand Z_h. Zu diesem Zeitpunkt war ein Segment der Dateneinheit mit der Sequenznummer k_h, für die

$$k_h \geq \mathtt{cRq} - \mathtt{sc} - \mathtt{mb} \cdot \mathtt{tr} - 1 \tag{5.4}$$

gilt, bereits gesendet worden. Zwischen Z_h und Z wurden nämlich nur höchstens `tr`-viele Daten-PDUs gesendet. Da in einer Daten-PDU nur höchstens `mb`-viele Segmente gesendet werden können und eine Dateneinheit immer aus mindestens einem Segment besteht, können höchstens $\mathtt{mb} \cdot \mathtt{tr}$-viele neue Dateneinheiten zwischen Z_h und Z gesendet worden sein, da nach APM *SSendInOrder* neue Dateneinheiten immer in inkrementeller Reihenfolge übertragen werden müssen. Dann folgt die Formel 5.4 aus der Formel 5.3.

Nach Satz 5.1 waren zum Zeitpunkt Z_h somit alle Dateneinheiten mit Sequenznummern a bestätigt, für die

$$a \geq \mathtt{cRq} - 2 \cdot \mathtt{sc} - \mathtt{rcc} - \mathtt{rcu} - \mathtt{mb} \cdot \mathtt{tr} - 1 \tag{5.5}$$

gilt. Da Station R nur Segmente von Dateneinheiten bestätigen darf, deren Sequenznummern kleiner als `maxIn` sind oder die im Empfangspuffer `rbk` gespeichert wurden, ist `maxIn` aufgrund der endlichen Kapazität von `rbk` zum Zeitpunkt Z_h größer als a_h:

$$a_h \triangleq \mathtt{cRq} - 2 \cdot \mathtt{sc} - 2 \cdot \mathtt{rcc} - 2 \cdot \mathtt{rcu} - \mathtt{mb} \cdot \mathtt{tr} - 1 \tag{5.6}$$

[19] Derartige Systemzustände kann man in TLA unter Verwendung von Vergangenheitsvariablen (siehe Abschnitt 3.3.4) spezifizieren.

Ab dem Zeitpunkt Z_h wurden von R nur noch Bestätigungsmitteilungen gesendet, in denen die durch `sacko` und `sackm` referenzierten Felder nur Segmentkennungen bzw. Sequenznummern enthalten, die mindestens den Wert a_h besitzen (APM *RAcknowledgeFA*). Da zum aktuellen Zeitpunkt Z jedoch nur PDUs ausgeliefert werden dürfen, die nach dem Zeitpunkt Z_h gesendet wurden, folgt die Behauptung aus der Formel 5.6. □

Als nächstes zeigen wir die Existenz der oberen Schranke M_s:

Satz 5.3 Sys *sei aus je einer Instanz der APMs* SBufferKey, RBufferKey, SCapacity, SAcknowledgeFA, RAcknowledgeFA *und* SSendInOrder *und der AMCs* MSDUId, ASDUId, MCorruptions, ACorruptions, MReorderings, MCapacity *und* AReorderingsMRq *komponiert.*

Dann ist

$$M_s \triangleq -1$$

eine obere Schranke der aktiven Dateneinheiten in der Instanz S.

`cRq` zeigt auf das Inkrement der Sequenznummer der höchsten bisher gesendeten Dateneinheit. Wie man mit einem einfachen Invariantenbeweis zeigen kann, der dem Leser überlassen bleibt, sind in allen Datenspeichern in den Instanzen S und R sowie den Basiskanälen M und A nur Sequenznummern gespeichert, die kleiner als `cRq` sind. Somit ist $\texttt{cRq} - 1$ trivialerweise eine Obergrenze für aktive Dateneinheiten. □

Aus den Sätzen 5.2 und 5.3 folgt, daß alle aktiven Nutzdatenkennungen k in Instanz S Sequenznummern aus dem Intervall

$$[\texttt{cRq} - 2 \cdot \texttt{sc} - 2 \cdot \texttt{rcu} - 2 \cdot \texttt{rcc} - \texttt{mb} \cdot \texttt{tr} - 1, \texttt{cRq} - 1] \tag{5.7}$$

besitzen.

Für die Empfangsinstanz R können wir in Abhängigkeit der Variablen `maxIn` ebenfalls eine obere Schranke M_r und eine untere Schranke m_r für aktive Dateneinheiten angeben, so daß sich alle aktiven Dateneinheiten im Intervall $[\texttt{maxIn} - m_r, \texttt{maxIn} + M_r]$ befinden. Die Existenz dieser Schranken beweisen wir in den Sätzen 5.4 und 5.5:

Satz 5.4 Sys *sei aus je einer Instanz der APMs* SBufferKey, RBufferKey, SCapacity, SAcknowledgeFA, RAcknowledgeFA *und* SSendInOrder *und der AMCs* MSDUId, ASDUId, MCorruptions, ACorruptions, MReorderings, MCapacity *und* AReorderingsMRq *komponiert.*

Dann ist

$$m_r \triangleq \texttt{sc} + \texttt{rcu} + \texttt{rcc} + \texttt{mb} \cdot (\texttt{c} + \texttt{tr}) + 1$$

eine untere Schranke für die aktiven Dateneinheiten in Instanz R.

Beweis: Außer dem aktuellen Systemzustand Z wird wie im Beweis von Satz 5.2 ein Systemzustand Z_h in der Vergangenheit betrachtet. Z_h galt direkt vor der `cMRq - c - tr`–ten Ausführung der Aktion `MRq`.

Aus dem APM *RBufferKey* folgt, daß die Dateneinheit $\texttt{maxIn} - 1$ beim Dienstnehmer ausgeliefert wurde (Aktion `In`). Deshalb wurde auch ein Segment dieser Dateneinheit von S über Kanal M übertragen (Aktion `MRq`). Nach den Anforderungen an das Basismedium M dürfen nur PDUs ausgeliefert werden, wenn seit ihrer Sendung höchstens $\texttt{c} + \texttt{tr}$-viele andere PDUs gesendet wurden (AMCs *SCapacity* und *SReorderings*). Im Systemzustand Z_h

war aufgrund der inkrementellen Sendereihenfolge (APM *SSendInOrder*) ein Segment der Dateneinheit l_h mit

$$l_h \geq \texttt{maxIn} - \texttt{mb} \cdot (\texttt{c} + \texttt{tr}) - 1 \tag{5.8}$$

bereits gesendet (siehe Formel 5.4). Nach Satz 5.1 waren somit im Zustand Z_h nur Dateneinheiten mit Sequenznummern a noch nicht bestätigt, für die

$$a \geq \texttt{maxIn} - \texttt{sc} - \texttt{rcu} - \texttt{rcc} - \texttt{mb} \cdot (\texttt{c} + \texttt{tr}) - 1 \tag{5.9}$$

gilt. Nach APM $SAcknowledgeFA$ wurden aber ab diesem Zeitpunkt nur noch Segmente unbestätigter Nutzdaten von Instanz S gesendet. Da im Zustand Z nur noch PDUs ausgeliefert werden dürfen, die später als zum Zeitpunkt Z_h gesendet wurden, folgt die Behauptung aus der Formel 5.9. □

Satz 5.5 Sys *sei aus je einer Instanz der APMs* SBufferKey, RBufferKey, SCapacity, SAcknowledgeFA, RAcknowledgeFA *und* SSendInOrder *und der AMCs* MSDUId, ASDUId, MCorruptions, ACorruptions, MReorderings, MCapacity *und* AReorderingsMRq *komponiert.*

Dann ist

$$M_r \triangleq \texttt{sc} + \texttt{rcu} + \texttt{rcc} - 1$$

eine obere Schranke für aktive Dateneinheiten in Instanz R.

Beweis: Im Empfangspuffer `rbk` können Segmente nur von maximal `rcu` + `rcc`-vielen Dateneinheiten enthalten sein, deren Kennung größer als `maxIn` ist (APM *RBufferKey*). Da alle Segmente, die durch R bestätigt werden, obwohl die Sequenznummern der zugehörigen Dateneinheiten größer oder gleich `maxIn` sind, im Empfangspuffer `rbk` gespeichert sein müssen, können nur `maxIn` + `rcu` + `rcc`-viele Dateneinheiten beim Sender bestätigt sein. Aufgrund der endlichen Kapazität des Sendepuffers `sbk` (APM *SCapacity*) folgt daraus

$$\texttt{cRq} \leq \texttt{maxIn} + \texttt{sc} + \texttt{rcu} + \texttt{rcc} \tag{5.10}$$

Da jedoch nur Dateneinheiten mit Sequenznummern kleiner als `cRq` in Instanz R oder Basiskanal M enthalten sein können (siehe Satz 5.3), gilt die Behauptung. □

Aus den Sätzen 5.4 und 5.5 folgt, daß

$$[\texttt{maxIn} - \texttt{sc} - \texttt{rcu} - \texttt{rcc} - \texttt{mb} \cdot \texttt{tr} - 1, \texttt{maxIn} + \texttt{sc} + \texttt{rcu} + \texttt{rcc} - 1] \tag{5.11}$$

das Intervall der in der Empfängerinstanz R aktiven Sequenznummern ist.

Die maximale Anzahl an Sequenznummern aktiver Dateneinheiten in der Sendeinstanz I_S und in der Empfangsinstanz I_R beträgt

$$I_S \triangleq 2 \cdot \texttt{sc} + 2 \cdot \texttt{rcu} + 2 \cdot \texttt{rcc} + \texttt{mb} \cdot \texttt{tr} + 1 \tag{5.12}$$

$$I_R \triangleq 2 \cdot \texttt{sc} + 2 \cdot \texttt{rcu} + 2 \cdot \texttt{rcc} + \texttt{mb} \cdot (\texttt{c} + \texttt{tr}) + 1 \tag{5.13}$$

Somit ist der erste Schritt der Entwicklung der FAPMs abgeschlossen, da I_R eine obere Schranke für die Anzahl aktiver Dateneinheiten in S bzw. R ist.

Im zweiten Schritt skizzieren wir die Entwicklung von APMs, die zu den in den Abschnitten 5.3.1, 5.3.2 und 5.3.3 vorgestellten APMs äquivalent sind, jedoch mit Ausnahme von Historienvariablen nur Variablen mit endlichen Wertebereichen besitzen.

Zunächst führen wir Sequenznummern für aktive Dateneinheiten ein. Aufgrund der Existenz der Schranke I_R reicht es aus, einer aktiven Dateneinheit anstatt einer Sequenznummer N_{nat} aus der Menge `key` eine Sequenznummer N_{mod} aus der Menge

$$\texttt{fkey} \triangleq \{0, \ldots, I_R - 1\} \cup \{\texttt{'notsent'}\} \tag{5.14}$$

zuzuweisen, um sie von allen anderen aktiven Dateneinheiten unterscheiden zu können. Die Abhängigkeit zwischen N_{nat} und N_{mod} wird durch die drei folgenden Zustandsabbildungen charakterisiert[20]:

$$N_{mod} = N_{nat} \; mod \; I_R \tag{5.15}$$

$$N_{nat} = (N_{mod} - \texttt{cRq}) \; mod \; I_R + \texttt{cRq} - I_R \tag{5.16}$$

$$N_{nat} = (N_{mod} - \texttt{maxIn} + m_r) \; mod \; I_R + \texttt{maxIn} - m_r \tag{5.17}$$

Durch Verwendung dieser Abbildungen können wir die Variablen `sbk`, `sbu`, `sba`, `sbr`, `rbk` und `rbu` durch Variablen des Typs `fpkey` ersetzen, ohne das Systemverhalten zu verändern. `fpkey` ist eine endliche Menge an Segmentkennungen. Die Anzahl der Segmente, in die Dateneinheiten zerlegt werden dürfen, ist beschränkt, so daß die Funktionen `skm` und `skn` (siehe Abschnitt 5.3.1) endliche Wertemengen haben. `skk` referenziert Sequenznummern aus der Menge `fkey`. Bei der Inkrementierung von `cRq` (Aktion `Rq`) bzw. `maxIn` (Aktion `In`) werden aus den oben genannten Puffern alle Segmente entfernt, deren Dateneinheiten Sequenznummern aus den Intervallen $[\texttt{cRq} - m_s, \texttt{cRq'} - m_s - 1]$ bzw. $[\texttt{maxIn} - m_r, \texttt{maxIn'} - m_r - 1]$ besitzen[21]. Diese Dateneinheiten sind nämlich durch das Inkrementieren inaktiv geworden. Die Variable `rd` in APM *RDuplicates*, die die Anzahl der Auslieferungen einer Dateneinheit beschreibt, wird für diese Sequenznummern auf den Wert 0 zurückgesetzt. Als nächstes zeigen wir, daß man Sequenznummern N_{mod} miteinander paarweise vergleichen kann, ohne sie zuvor in Sequenznummern N_{nat} überführen zu müssen. Dazu verwenden wir die von Tanenbaum in [Tan89] eingeführte boolesche Funktion

```
between (x, y, z : fkey) ≙
  (x ≤ y ∧ y < z) ∨
  (z < x ∧ x ≤ y) ∨
  (y < z ∧ z < x);
```

`between` ist wahr, wenn sich der Wert y „zwischen“ x und z befindet. Aus den Formeln 5.16 und 5.17 kann man für zwei aktive Dateneinheiten mit Sequenznummern x und y aus `key` die Eigenschaften

$$x < y \Leftrightarrow \texttt{between}(\texttt{cRq} - m_s, x \; mod \; I_R, y \; mod \; I_R) \tag{5.18}$$

$$x < y \Leftrightarrow \texttt{between}(\texttt{maxIn}, x \; mod \; I_R, y \; mod \; I_R) \tag{5.19}$$

herleiten.

Die Zähler `cRq` und `maxIn` können wir durch Zähler `fcRq` und `fmaxIn` aus `fkey` mit

$$\texttt{fcRq} \triangleq \texttt{cRq} \; mod \; I_R$$

$$\texttt{fmaxIn} \triangleq \texttt{maxIn} \; mod \; I_R$$

[20] Immer, wenn N_{mod} den speziellen Wert `'notsent'` besitzt, hat N_{nat} ebenfalls den Wert `'notsent'`.

[21] `cRq'` und `maxIn'` bezeichnen die Werte von `cRq` und `maxIn` nach Ausführung der Aktionen `Rq` bzw. `In`.

ersetzen, die zur Identifikation der als nächstes zu sendenden bzw. auszuliefernden Dateneinheit genügen. Wie die oben angeführten Datensegmentspeicher werden auch `cRq` und `maxIn` in den abgeleiteten APMs als Historienvariablen gespeichert, da sie zur Angabe von Refinement Mappings (siehe Abschnitt 3.3.4) zwischen den abgeleiteten APMs und ihren Originalen notwendig sind.

Die einzigen Strukturen aus den Datentransfer-APMs, für die wir noch keine Abbildung auf endliche Kontrollstrukturen angegeben haben, sind `scred` und `rcred` aus den APMs *SCredit* und *RCredit*. Man kann diese Variablen durch

$$\begin{aligned} \texttt{fscred} &\triangleq \texttt{scred}\ mod\ I_R \\ \texttt{frcred} &\triangleq \texttt{rcred}\ mod\ I_R \end{aligned}$$

ersetzen, obwohl sie damit unter Umständen auf inaktive Dateneinheiten verweisen. Man kann leicht beweisen, daß `rcred` Werte zwischen $\texttt{maxIn} - 1$ und $\texttt{maxIn} + \texttt{rcu} + \texttt{rcc} + mo$ annehmen kann. Der Prozeßtypparameter mo gibt eine Obergrenze für die Anzahl an Nutzdaten an, für die der Empfänger Kredit geben darf, obwohl sie eventuell die Pufferkapazität übersteigen. Da es nicht sinnvoll ist, einen Kredit zu geben, der die Kapazität des Sendepuffers übersteigt, nehmen wir im folgenden $mo \leq \texttt{sc}$ an. Dann besitzt `rcred` immer nur Werte aktiver Nutzdatenkennungen. Im Gegensatz dazu kann `scred` jedoch auch Werte außerhalb des Intervalls der in Instanz S aktiven Dateneinheiten annehmen. Es gilt nämlich immer

$$\texttt{scred} \in [\texttt{cRq} - \texttt{sc}, \texttt{cRq} + \texttt{sc} + \texttt{rcu} + \texttt{rcc}]$$

Werte von `scred`, die größer oder gleich `cRq` sind, werden demnach durch die Abbildung 5.16 falsch übersetzt. Das beeinflußt das Systemverhalten jedoch nicht, da alle falsch referenzierten Sequenznummern auf Dateneinheiten verweisen, die schon bestätigt sind (siehe Satz 5.1). Nach APM *SAcknowledgeFA* dürfen bestätigte Dateneinheiten nämlich nicht mehr über Kanal S übertragen werden.

Die letzten Änderungen betreffen die über die Basiskanäle M und A übertragenen PDUs, die Sequenznummern und Segmentkennungen aus einem endlichen Wertebereich enthalten. `skey`, `sacko` und `snak` haben `fpkey` als Wertemenge, während `scred` und `sackm` auf Elemente aus der Menge `fkey` verweisen.

Wir können nun aus den auf diese Weise umgeformten APMs die FAPMs herleiten. Dazu lassen wir in den Spezifikationen die Historienvariablen weg, da sie das Verhalten der FAPMs nicht beeinflussen. Aus Gründen des besseren Verständnisses der Spezifikationen wurden zudem für die Variablen die gleichen Bezeichner wie in den APMs des Frameworks verwendet (z.B. `cRq` anstatt `fcRq`). Als Beispiel wird zur Erläuterung das FAPM *SBufferKey* angegeben:

```
PROCESS SBufferKey    Sequenznummernverwaltung an der Sendeinstanz S
                    ( pdu, pci, usd : Any CONSTANT ;  PDUs, PCIs und Nutzdaten
                      encpdu : SUBSET (pdu) ;         Menge unverfaelschter PDUs
                      spci : SET[pdu → pci] ;         Zugriff auf die PCI einer PDU
                      skey : SET[pci → (SUBSET (fpkey) ∪ {"≪ ≫"})] ;
                        Zugriff auf die Sequenznummern uebertragener Nutzdaten in einer PCI
                      mseg : nat;                     Max. Anzahl an Segmenten
                      skk : SET[fpkey → fkey] ;
                        Sequenznummer der Dateneinheit, zu der das Segment gehoert
                      skn : SET[fpkey → {k | 0 ≤ k < mseg}] ; Interne Segmentnummer
```

```
                    skm : SET[fpkey → {k | 0 < k ≤ mseg}] ;
                      Anzahl an Segmenten einer Dateneinheit
                    usdsize : SET[usd → {k | 0 < k ≤ mseg}] ;
                      Groesse von Dateneinheiten (wieviele Segmente sind notwendig ?)
                    mb : Nat ;                Max. Anzahl an Segmenten in einer PDU
                    ir : Nat ;                Anzahl an Sequenznummern
                    ms : nat )                Anzahl aktiver Sequenznummern
IMPORT Constants;
BODY
VARIABLES
  cRq : fkey;             Sequenznummernzaehler
  sbk : SUBSET (fpkey); Menge verwendeter Segmentkennungen
INIT ≙ cRq = 0 ∧ sbk = {};
ACTIONS
  Rq (krq : fkey; d : usd) ≙ Uebergabe vom Dienstnehmer
    krq = cRq ∧
    sbk' = (sbk \ { k | skk[k] = (cRq - ms) mod ir}) ∪
              { k | k ∈ fpkey : skk[k] = krq ∧ skn[k] < usdsize[d] ∧
                                    skm[k] = usdsize[d] } ∧
    cRq' = inc(cRq);
  MRq ( krq : ((SUBSET(fpkey)) ∪ {"≪≫"}); p : pdu ) ≙
    Sendung ueber Basiskanal M
    p ∈ encpdu ∧ krq = skey[spci[p] ] ∧
    ( krq = "≪≫" ∨ ( krq ⊆ sbk ∧ card(krq) ≤ mb ) ) ∧
    cRq' = cRq ∧ sbk' = sbk;
END
```

Anstatt der Mengen `key` und `pkey` im APM *SBufferKey* werden beim FAPM die endlichen Mengen `fkey` der Nutzdatenkennungen (siehe Formel 5.14) und `fpkey` der Segmentdatenkennungen verwendet. Die Variablen `cRq` und `sbk` im FAPM entsprechen den oben eingeführten Variablen `fcRq` und `fsbk`. `cRq` nimmt bei Ausführung der Aktion `Rq` den Wert (`cRq`+1) *mod* `ir` an, wobei `ir` für die Anzahl aktiver Dateneinheiten I_R steht. Wenn `Rq` schaltet, werden aus `sbk` immer die Segmentkennungen der ältesten aktiven Dateneinheit `cRq` $- m_s$ gelöscht.

Die AMCs müssen nicht in Systeme mit endlichen Zustandsräumen umgewandelt werden, da sie nur abstrakte Beschreibungen der Schnittstelle des untergeordneten Dienstes sind und im Gegensatz zu Protokollspezifikationen nicht implementiert werden.

5.5.2 Entwurf der FAPMs der Interconnection-Coordination

In den SCs und APMs, die in den Abschnitten 5.2.6 und 5.3.4 vorgestellt wurden, dienen die Verbindungskennungen `no` zur Referenzierung einzelner Verbindungen in einem Feld von Verbindungsspezifikationen, durch das ein Transportsystem modelliert wird. In den FAPMs müssen wir `no` durch Verbindungskennungen aus einer endlichen Menge ersetzen. Zum einen besitzt nämlich das APM *OpenConnections* die Variable `opcon` mit einem unendlichen Wertebereich. Zum anderen müssen wir berücksichtigen, daß die Anzahl der virtuellen Basiskanäle M und A in dem Feld von Verbindungsspezifikationen unendlich ist. Um diese virtuellen Kanäle durch eine endliche Anzahl realer Kanäle zu implementieren, erhalten alle PDUs eines virtuellen Kanals eine eindeutige Kennung der zugehörigen Verbindung. Aufgrund der endlichen Größe der PCI einer PDU kann man jedoch nur endlich viele Verbindungskennungen speichern.

Die Überführung der APMs in die FAPMs nutzt die Eigenschaft von Rechnernetzen aus, daß zu jedem Zeitpunkt nur eine endliche Anzahl an Verbindungen aktiv ist, die durch eindeutige Kennungen identifiziert werden können. Zwar ist diese Eigenschaft zu der im vorangegangenen Abschnitt untersuchten Eigenschaft analog, daß in einer Verbindung gleichzeitig nur endlich viele Dateneinheiten aktiv sind, doch müssen wir berücksichtigen, daß die Lebenszeit von Verbindungen (und damit von Verbindungskennungen) im Gegensatz zu der von Sequenznummern sehr unterschiedlich sein kann[22]. Deshalb kann man hier kein Intervall aktiver Verbindungskennungen mit einer festen Intervallänge angeben.

Wir lösen das Problem, indem eine aktive Verbindungskennungsverwaltung spezifiziert wird. Jeder Knoten eines Rechnernetzes verfügt über eine endliche Menge `cno` an Kennungen. Beim Aufbau einer neuen Verbindung wird dieser eine Kennung aus `cno` zugewiesen, durch die sie während ihrer Lebenszeit eindeutig referenziert wird. Nach dem Verbindungsende wird die Kennung dem Knoten zurückgegeben und kann einer anderen Verbindung zugeordnet werden. Das Problem dieser Methode liegt darin, daß man unter Umständen das endgültige Verbindungsende nicht terminieren kann, wenn beim Verbindungsabbau Übertragungsfehler aufgetreten sind. In den meisten Protokollen wird dieses Problem entweder nicht beachtet oder wie in den Protokollen PATROCLOS ([Bra93]) und XTP ([Pro89, XTP95]) durch zeitweise „Quarantäne“ der Verbindungskennungen gelöst. Darunter versteht man, daß eine Verbindungskennung solange keiner neuen Verbindung zugeordnet wird, bis ein Empfang von PDUs aus der alten Verbindung ausgeschlossen ist. Wenn eine Übertragungsverzögerung pd des untergeordneten Mediums und eine maximale Verzögerungszeit d durch den Kommunikationspartner bekannt ist, kann man die Sperrzeit durch $2 \cdot pd + d$ begrenzen. Eine weitere Methode zur Feststellung des Verbindungsendes ist ein Datenaustausch mit dem Kommunikationspartner vor der Wiedervergabe der Verbindungskennung, durch den ermittelt wird, ob die Verbindung aus der Sicht des Partners endgültig abgeschlossen ist. Uns ist jedoch kein Protokoll bekannt, das Kennungen auf diese Weise verwaltet. Wir haben im Framework deshalb nur FAPMs eingefügt, die die „Quarantäne“-Methode modellieren.

Eine Verbindung ist erst vollständig abgeschlossen, wenn die folgenden Bedingungen gelten:

1. Der Zustand der beiden Dienstzugangspunkte (APM *ConnectionState*) ist `'finished'`. Daten können demnach nicht mehr über die Transportdienstschnittstelle ausgeliefert werden.

2. In den Basiskanälen M und A der Verbindung sind keine PDUs mehr vorhanden, die noch an eine der Protokollinstanzen ausgeliefert werden können.

3. Die Protokollinstanzen müssen nicht mehr auf eingetroffene PDUs reagieren.

Nur wenn diese Bedingungen gelten, kann man die Kennung einer Verbindung ohne die Gefahr eines Protokollfehlers freigeben und einer neuen Verbindung zuordnen.

Wir führen die vier neuen APMs *AssignCNo*, *SelectCNo*, *FreeCNo* und *LiveFreeCNo* ein, durch die wir die Verwaltung der Verbindungskennungen aus der Menge `cno` in einem Rechnerknoten modellieren. Die Verwaltung von Verbindungskennungen innerhalb einer Verbindung wird durch das FAPM *AssignCNo* beschrieben. Die Spezifikation berücksichtigt, daß in einigen Protokollen (z.B. XTP) aus dem Grund, ankommende PDUs schnell den zugehörigen

[22] Einzelne Verbindungen können sogar ständig aktiv sein.

Verbindungskontexten zuordnen zu können, Verbindungen durch zwei separate Kennungen identifiziert werden, die im Kontext jeweils einer der beiden Rechnernetzknoten eindeutig sind. *AssignCNo* besitzt deshalb zwei Variablen `myn` und `pn`, in denen die Kennungen gespeichert werden. Die in der eigenen Station eindeutige Kennung wird bei der Sendung (Aktion `SRq`) bzw. Auslieferung (Aktion `SIn`) einer Signalisierungsdateneinheit der Variablen `myn` zugewiesen. Die Speicherung der in der Partnerstation eindeutigen Kennung in der Variablen `pn` erfolgt beim Empfang einer PDU. In aktiven Verbindungen ist mindestens eine der beiden Kennungen gespeichert, die der PCI jeder gesendeten PDU hinzugefügt wird. Eine PDU wird nur empfangen, wenn entweder noch keine Kennungen vergeben wurden, oder die in der PCI und in der empfangenden Instanz gespeicherten Kennungen miteinander übereinstimmen.

Das FAPM *SelectCNo* spezifiziert die Verwaltung der Kennungen in einer Station. Verlangt eine Protokollinstanz einer Verbindung die Zuweisung einer neuen Kennung, so wird eine noch nicht vergebene Kennung aus der Menge `mycno` der zur Verfügung stehenden Kennungen vergeben. Die Kooperation der Protokollinstanz und der Kennungsverwaltung modellieren wir durch die Aktionsparameter `newmyn` und `gnm` in den Aktionen `SRq` und `SIn`. `newmyn` enthält die zugewiesene Verbindungskennung, während die Protokollinstanz durch Setzen des Flags `gnm` auf `'yes'` bzw. `'no'` angibt, ob sie die Zugabe einer neuen Kennung wünscht. In *SelectCNo* werden Kennungen erst nach Ablauf der „Quarantäne“ wieder freigegeben. Die Aktion `Free` modelliert die Freigabe einer Kennung.

Die Einhaltung der drei oben genannten Bedingungen bei der Freigabe von Verbindungskennungen wird durch das FAPM *FreeCNo* modelliert. Danach darf sowohl eine im eigenen Kontext als auch im Kontext des Partners eindeutige Kennung erst freigegeben werden, wenn weder eine Aktion an der Transportschnittstelle noch an der Schnittstelle zu den Basiskanälen mehr schalten darf. Die Garantie, daß die Kennungen abgeschlossener Verbindungen irgendwann freigegeben werden, wird durch das FAPM *LiveFreeCNo* beschrieben, in dem die mit `Free` gekoppelte Aktion `fFree` schwache Fairness besitzt.

Das APM *OpenConnections* kann nun in ein FAPM überführt werden, indem man in der Variablen `opcon` die an Verbindungen zugewiesenen Kennungen aus `cno` speichert. Das APM *FindDestination* kann man dagegen direkt in ein FAPM überführen.

Die FAPMs *AssignCNo*, *FreeCNo* und *LiveFreeCNo* beschreiben das Verhalten einer einzigen Protokollinstanz in einer Verbindung. Deshalb werden jeweils zwei Instanzen dieser FAPMs zu einer Verbindung komponiert, bevor sie durch die cTLA$^+$-Prozeßfeldoperation in ein Feld von Verbindungsspezifikationen überführt wird. Die FAPMs *SelectCNo*, *OpenConnections* und *FindDestination* modellieren dagegen das Verhalten von Rechnernetzknoten. Sie werden mit dem Feld von Verbindungsspezifikationen komponiert.

In den FAPMs wurden die Prozeßtypparameter `cona`, `conb` (FAPMs *SelectCNo*, *OpenConnections*), `con` und `adrlist` (FAPM *FindDestination*) beibehalten, obwohl sie unendlich große Mengen von Verbindungskennungen `no` beschreiben. Sie modellieren die Topologie eines Rechnernetzes, indem sie angeben, welche Verbindungen einem bestimmten Knoten zugeordnet sind. Wir nehmen an, daß jeder Knoten die ihm bzw. seinen Partnerstationen zugeordneten Verbindungen effizient ermitteln kann, ohne die Verbindungskennungen `no` speichern zu müssen. Die Ermittlung geeigneter Verbindungen wird in der Realität durch den Vermittlungsdienst erbracht, den wir durch die Prozeßtypparameter `cona`, `conb`, `con` und `adrlist` abstrakt modellieren.

5.6 Theoreme zu den SCs

Die an dieser Stelle vorgestellten Theoreme dienen dem Beweis, daß eine aus APMs und AMCs zusammengesetzte abstrakte Protokollspezifikation eine aus SCs komponierte Dienstspezifikation implementiert. Der Aufbau der Theoreme wurde in Abschnitt 5.1.4 beschrieben. Ihre Korrektheit wurde mit Hilfe von TLA-Verifikationen bewiesen (siehe Abschnitt 3.3.3). Wir veranschaulichen das Vorgehen bei der Verifikation der Theoreme anhand je eines Beispiels einer Safetyverifikation und eines Livenessbeweises. In den Abschnitten 5.6.1 und 5.6.2 werden die Theoreme zum Beweis der Safety- und Liveness-SCs des Datentransfers erläutert. Theoreme zur Verifikation der SCs der Intraconnection-Coordination sind in Abschnitt 5.6.3 beschrieben. Da die SCs und die APMs der Interconnection-Coordination gleichartig aufgebaut sind, sind die Theoreme trivial und werden nicht explizit erläutert.

5.6.1 Theoreme zu den Safety-SCs des Datentransfers

Wir stellen zunächst ein Theorem zum Beweis des SCs *SDUId* vor (siehe Abschnitt 5.2.1) und zeigen anschließend seine Gültigkeit:

```
LET Pars ≜  Anforderungen an die Prozesstypparameter
            mtc ⊆ { (p, q) | q ∉ encpdu ∨
                             skey[spci[p] ] = skey[spci[q] ] };
    Sys ≜   Subsystem aus APMs und AMCs, durch das SDUId erfuellt wird
            SBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn, skm,
                        usdsize, mb) ∧
            RBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn, skm,
                        usdsize, rcu, rcc) ∧
            MSDUId ∧
            MCorruptions (pdu, mtc) ∧
            MPhantoms (pdu,encpdu) ∧
            CC_SDUId;
IN Pars ∧ Sys ⇒ SDUId     Theorem
```

`Sys` gibt das Subsystem der Protokollspezifikation an, das aus den APMs *SBufferKey* und *RBufferKey* sowie der AMCs *MSDUId*, *MCorruptions* und *MPhantoms* komponiert ist. Die geforderten Abhängigkeiten zwischen den Prozeßtypparametern werden zum einen durch Gleichheit von Bezeichnern (z.B. `encpdu`) und zum anderen durch die Bedingung `Pars` modelliert. Im oben aufgeführten Theorem wird durch `Pars` beschrieben, daß bei der Übertragung von PDUs über den Basiskanal M Datenverfälschungen nur toleriert werden dürfen, wenn sie entweder vom Empfänger erkannt werden können (`q` $\notin$ `encpdu`) oder wenn zumindest die übertragenen Segmentkennungen nicht verfälscht sind (`skey[spci[p]] = skey[spci[q]]`).

Die Kopplung der Aktionen in `Sys` wird durch die Bedingung $\mathtt{CC}_{\mathtt{SDUId}}$ beschrieben (siehe Abschnitt 4.1.2), deren Aufbau aus Gründen der Übersicht jedoch nicht explizit aufgeführt ist. Die Aktionen mit denselben Bezeichnern werden miteinander gekoppelt[23]. In unserem Beispielsystem `Sys` werden die Aktionen `MRq` in *SBufferKey*, *MSDUId* und *MCorruptions* sowie die Aktionen `MIn` in *RBufferKey*, *MSDUId*, *MCorruptions* und *MPhantoms* miteinander gekoppelt. Außerdem wird vollständiges Interleaving für alle nicht in einem Prozeßtyp gekoppelten Aktionen angenommen. Man kann `Sys` also durch die cTLA$^+$-Operation der Prozeßkomposition (siehe Abschnitt 4.3.2) aus einzelnen APMs und AMCs bilden:

[23] Ausnahmen sind in den Kommentaren zu den Theoremen aufgeführt.

```
PROCESS Sys    System, das das SC SDUId realisiert
            ( pdu, pci, usd : Any CONSTANT ;  PDUs, PCIs und Nutzdaten
              encpdu : SUBSET(pdu );          Menge unverfaelschter PDUs
              spci : SET[ pdu → pci] ;        Zugriff auf die PCI einer PDU
              skey : SET[ pci → (SUBSET( pkey) ∪ {"≪ ≫"})] ;
                        Zugriff auf die Sequenznummern uebertragener Nutzdaten in einer PCI
              skk : SET[ pkey → key] ;
                        Sequenznummer der Dateneinheit, zu der das Segment gehoert
              skn : SET[ pkey → Nat] ;        Interne Segmentnummer
              skm : SET[ pkey → Nat] ;        Anzahl an Segmenten einer Dateneinheit
              usdsize : SET[ usd → Nat] ;
                        Groesse von Dateneinheiten (wieviele Segmente sind notwendig?)
              mb  : Nat ∪ {"infty"} ;   Maximale Anzahl an Segmenten in einer PDU
              rcu : Nat ∪ {"infty"} ;   Groesse des
              rcc : Nat ;               Empfangspuffer
              mtc : SUBSET(pdu × pdu)) Menge erlaubter Verfaelschungen in Basiskanal M

PROCESSES  Subsystem
  SBK: SBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn, skm,
                   usdsize, mb);
  RBK: RBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn, skm,
                   usdsize, rcu, rcc);
  MS : MSDUId;
  MC : MCorruptions (pdu, mtc);
  MP : MPhantoms (pdu,encpdu);

ACTIONS    Aktionen
  Rq (krq : key; d : usd) ≙ Uebergabe vom Dienstnehmer
    SBK.Rq (krq,d) ∧ RBK.Stutter ∧ MS.Stutter ∧ MC.Stutter ∧ MP.Stutter;
  In (krq : key; d : usd) ≙ Auslieferung an den Dienstnehmer
    SBK.Stutter ∧ RBK.In (krq,d) ∧ MS.Stutter ∧ MC.Stutter ∧ MP.Stutter;
  MRq ( krq : (SUBSET( pkey) ∪ {"≪ ≫"}); p : pdu; mkrq : key ) ≙
    Sendung ueber Basiskanal M
    SBK.MRq(krq,p) ∧ RBK.Stutter ∧ MS.MRq(mkrq) ∧ MC.MRq (mkrq,p) ∧
    MP.Stutter;
  MIn ( krq : (SUBSET(pkey) ∪ {"≪ ≫"}); p : pdu; mkrq : key ) ≙
      Empfang von Basiskanal M
    SBK.Stutter ∧ RBK.MIn (krq,p) ∧ MS.MIn(mkrq) ∧ MC.MIn (mkrq,p) ∧
    MP.MIn (mkrq,p);
END
```

Wir zeigen im folgenden die Gültigkeit des Theorems:

Satz 5.6

$$\texttt{Pars} \wedge \texttt{Sys} \Rightarrow \texttt{SDUId}$$

Beweis: Die Verifikation erfolgt nach dem von Lamport vorgegebenen Schema für Safetybeweise (vgl. [Lam94], siehe auch Kapitel 3.3.3). Zunächst stellt man fest, daß die Invariante I_{SDUId} gültig ist:

$$
\begin{array}{l}
\texttt{I}_{\texttt{SDUId}} \triangleq \texttt{SBK.sbk} \subseteq \{\ \texttt{k} \in \texttt{pkey}\ \mid\ \texttt{skk[k]} < \texttt{SBK.cRq}\ \} \wedge \\
\quad \texttt{MC.MBuf} \subseteq \{\ [[\texttt{k,p}]]\ \mid\ \texttt{k} < \texttt{MS.cMRq} \wedge \texttt{p} \in \texttt{encpdu} \wedge \\
\qquad\qquad\qquad\qquad (\ \texttt{skey[spci[p]]} \subseteq \{\texttt{k} \in \texttt{pkey}\ \mid\ \texttt{skk[k]} < \texttt{SBK.cRq}\} \vee \\
\qquad\qquad\qquad\qquad\ \ \texttt{skey[spci[p]]} = \texttt{"}\ll\gg\texttt{"}\ )\ \} \wedge \\
\quad \texttt{RBK.rbk} \subseteq (\{\ \texttt{k} \in \texttt{pkey}\ \mid\ \texttt{skk[k]} < \texttt{SBK.cRq}\ \} \cup \{\texttt{"notsent"}\})
\end{array}
$$

Sie sagt aus, daß der Sendepuffer $SBK.sbk$, der Empfangspuffer $RBK.rbk$ und die im Basiskanal M befindlichen PDUs nur Segmente von Nutzdaten enthalten, deren Sequenznummern kleiner als der Zähler $SBK.cRq$ sind. Außerdem kann $RBK.rbk$ vom Empfänger erkannte Phantome enthalten (Sequenznummer `'notsent'`). Die Gültigkeit von I_{SDUId} wird bewiesen, indem wir zeigen, daß die Formeln

$$Init_{Sys} \Rightarrow I_{SDUId} \tag{5.20}$$

$$\pi \wedge I_{SDUId} \wedge \texttt{Pars} \Rightarrow I'_{SDUId} \tag{5.21}$$

wahr sind, was dem Leser überlassen bleibt. (Der Bezeichner π in Formel 5.21 steht für jede einzelne der Aktionen aus `Sys` (`Rq`, `In`, `MRq` und `MIn`).) Aufgrund der TLA-Regel INV1 kann daraus die Invariantenbedingung $\texttt{Sys} \Rightarrow \Box I_{SDUId}$ gefolgert werden. Wegen der TLA-Regel INV2 können wir dann alle Aktionen π in `Sys` durch

$$\pi \wedge I_{SDUId} \wedge I'_{SDUId} \tag{5.22}$$

ersetzen.

Das Theorem wird nach dem Schema der TLA-Verfeinerungsbeweise verifiziert (siehe Abschnitt 3.3.3). Wir ersetzen $SDUId$ durch eine Spezifikation $\overline{SDUId} \triangleq SDUId(\texttt{SDUId.cRq}/\texttt{SBK.cRq})$. Dadurch wird `cRq`, die einzige Variable in $SDUId$, durch die Variable `cRq` aus dem Prozeßtyp `SBK` ersetzt. Die Spezifikationen Sys und $\overline{SDUId}$ verwenden somit die gleichen Variablen. Im nächsten Schritt beweisen wir, daß für die Initialbedingung

$$Init_{Sys} \Rightarrow Init_{\overline{SDUId}} \tag{5.23}$$

gilt. Der Beweis ist trivial, da $Init_{Sys}$ als Konjunkt $\texttt{SBK.cRq} = 0$ besitzt, woraus sofort $\overline{\texttt{SDUId.cRq}} = 0$ folgt. Als nächstes beweisen wir für die Aktionen von `Sys` ($Inv \triangleq I_{SDUId} \wedge I'_{SDUId}$):

$$\forall krq, d : \texttt{Sys.Rq(krq,d)} \wedge Inv \Rightarrow \overline{\texttt{SDUId.Rq(krq)}} \tag{5.24}$$

$$\forall krq, p, mkrq : \texttt{Sys.MRq(krq,p,mkrq)} \wedge Inv \Rightarrow unchanged(\overline{\texttt{SDUId.cRq}}) \tag{5.25}$$

$$\forall krq, p, mkrq : \texttt{Sys.MIn(krq,p,mkrq)} \wedge Inv \Rightarrow unchanged(\overline{\texttt{SDUId.cRq}}) \tag{5.26}$$

$$\forall krq, d : \texttt{Sys.In(krq,d)} \wedge Inv \Rightarrow \overline{\texttt{SDUId.In(krq)}} \tag{5.27}$$

Der Beweis von Formel 5.24 ist trivial, da `SBK.cRq` sowohl in Aktion `Sys.Rq` als auch in Aktion $\overline{\texttt{SDUId.Rq}}$ inkrementiert wird. Die Beweise von 5.25 und 5.26 sind ebenfalls trivial, da die Variable `SBK.cRq` jeweils stottert. Etwas schwieriger ist der Beweis von Formel 5.27. Dort profitieren wir jedoch von der Invariante I_{SDUId}, durch die sichergestellt wird, daß alle Segmente im Empfangspuffer `RBK.rbk` Segmentkennungen bezeichnen, deren Nutzdaten entweder kleiner als `SBK.cRq` sind oder den Wert `'notsent'` haben. Daraus folgt die Gültigkeit der Aktion $\overline{\texttt{SDUId.In}}$.

SCs / APMs	SDU-Id	Corr-upt.	Dup-lic.	Reor-der.	Gaps	Phan-toms	Ca-pac.	Rel.-Corr.	Rel.-Gaps	Rel.-Phan.
SBufferKey	⋆	⋆						⋆	⋆	
SBufferUsd		⋆						⋆		
SCapacity							⋆			
SReject								⋆	⋆	
RBufferKey	⋆	⋆						⋆	⋆	⋆
RBufferUsd		⋆						⋆		⋆
RDuplicates			⋆							
RReorderings				⋆						
RGaps					⋆					
RPhantoms						⋆				
RAcknowledge							⋆			
RVolunt.Gaps									⋆	
AMCs										
MSDUId	⋆	⋆						⋆	⋆	⋆
MCorruptions	⋆	⋆						⋆	⋆	
MPhantoms	⋆	⋆						⋆	⋆	
MRel.Corrupt.								⋆	⋆	
MRel.Phan.								⋆		⋆
MRel.Gaps									⋆	
ASDUId							⋆			
ACorruptions							⋆			
APhantoms							⋆			

Tabelle 5.1: Theoreme zum Beweis der Safety-SCs des Datentransfers (Zusammensetzung von `Sys` aus APMs und AMCs)

Mit Hilfe der TLA-Regel STL4 (siehe Abbildung 3.2) kann man aus den Formeln 5.23 bis 5.27 die Gültigkeit des Theorems beweisen. □

In Tabelle 5.1 werden die Theoreme zum Beweis der Safety-SCs skizziert, indem jeweils die Zusammensetzung des Subsystems `Sys` aufgeführt wird. Zur Implementierung des SC *Corruptions* werden die APMs zur Verwaltung der Segmentkennungen und der Segmentdaten verwendet. Der Basiskanal M liefert nur verfälschte PDUs und Phantome aus, die vom Empfänger erkannt werden können ($\mathtt{p} \notin \mathtt{encpdu}$). Die SCs *Duplicates*, *Reorderings*, *Gaps* und *Phantoms* beschreiben Diensteigenschaften, die sich durch gleichartige Protokollmechanismen realisieren lassen. Der Bestätigungskanal A wird bei der Realisierung des SC *Capacity* verwendet. *Capacity* ist erfüllt, wenn die Kapazität des Dienstes (Prozeßtypparameter `c`) nicht größer als die Summe der Sendepuffergröße `sc` und der Empfangspuffergröße `rcc + rcu` ist.

Die SCs, die relative Übertragungsgüten modellieren, werden eingesetzt, wenn keine expliziten Fehlerbehebungsmaßnahmen verwendet werden und auch der Basiskanal M relative Übertragungsgüten garantiert. Somit ist die durch das SC *RelativeCorruptions* beschriebene relative Häufigkeit von Verfälschungen (Prozeßtypparameter `maxrc`) nicht größer als die Summe der relativen Häufigkeiten an Verfälschungen und Phantomen in M. Die relative Häufigkeit `maxrg` von Lücken im Datenstrom (SC *RelativeGaps*) ist nicht größer als die Summe der relativen Häufigkeiten der Verfälschungen und Lücken in Kanal M. Das Theorem zum Beweis des

SC *RelativePhantoms* zeigt, daß die relative Anzahl von Phantomen im Ausgabedatenstrom nicht höher als in Kanal M ist.

5.6.2 Theoreme zu den Liveness-SCs des Datentransfers

Die Livenesseigenschaften des Transportdienstes werden durch die SCs *LiveIn*, *LiveInNoAttr* und *LiveCf* modelliert (siehe Abschnitt 5.2.2). Zur Realisierung dieser Eigenschaften durch abstrakte Protokolle haben wir die APMs *SLiveMRq*, *RLiveARq* und *RLiveIn* eingeführt. Die Liveness der Basiskanäle M und A wird durch die AMCs *MLiveIn* und *ALiveIn* garantiert.

In Abschnitt 5.3.1 haben wir außer den für den Beweis der Safety-SCs notwendigen APMs eine Reihe weiterer APMs eingeführt. Die durch sie modellierten Protokollmechanismen stellen eine hohe Datenübertragungsrate sicher, indem die Sendung von PDUs, die für den Kommunikationsfortschritt nicht notwendig sind, eingeschränkt wird. Damit nicht versehentlich auch die Sendung für den Fortschritt wichtiger PDUs unterdrückt wird, werden diese APMs beim Entwurf und Beweis der Theoreme von Livenesseigenschaften berücksichtigt. Wir werden im folgenden für das SC *LiveInNoAttr* ein Theorem vorstellen und seine Gültigkeit beweisen:

```
LET Pars ≙ Anforderungen an die Prozesstypparameter
           mla = { (p,q) | skey[spci[p] ] = skey[spci[q] ] ∧
                           stack[spci[p] ] = stack[spci[q] ] ∧
                           stcre[spci[p] ] = stcre[spci[q] ] ∧
                           p ∈ encpdu ∧ q ∈ encpdu } ∧
           mtc ⊆ { (p, q) | q ∉ encpdu ∨
                            ( skey[spci[p] ] = skey[spci[q] ] ∧
                              stack[spci[p] ] = stack[spci[q] ] ∧
                              stcre[spci[p] ] = stcre[spci[q] ] ) };
           ala = { (p,q) | sack[spci[p] ] = sack[spci[q] ] ∧
                           snak[spci[p] ] = snak[spci[q] ] ∧
                           scre[spci[p] ] = scre[spci[q] ] ∧
                           p ∈ encpdu ∧ q ∈ encpdu } ∧
           atc ⊆ { (p, q) | q ∉ encpdu ∨
                            ( sack[spci[p] ] = sack[spci[q] ] ∧
                              snak[spci[p] ] = snak[spci[q] ] ∧
                              scre[spci[p] ] = scre[spci[q] ] ) };
    Sys ≙ Subsystem aus APMs und AMCs, durch das LiveInNoAttr erfuellt wird
           SLiveMRq (pdu, pci, usd, encpdu, spci, skey, sack, snak, scre,
                     stack, stcre, skk, skn, skm, usdsize, usdsplit, mcr, rcc) ∧
           RLiveARq (pdu, pci, usd, encpdu, spci, skey, sack, snak, scre,
                     stack, stcre, skk, skn, skm, usdsize, rcu, rcc, ma, mr,
                     mo) ∧
           RLiveIn (usd, skk, skn, skm, usdsize, usdsplit, rcu, rcc, tg) ∧
           RAcknowledge (pdu, pci, encpdu, spci, sack, skk, rcu, rcc, ma) ∧
           MSDUId ∧
           MCorruptions (pdu, mtc) ∧
           MPhantoms (pdu, encpdu) ∧
           MLiveIn (pdu, mmaxrg, mtg, mc, mla) ∧
           ASDUId ∧
           ACorruptions (pdu, atc) ∧
           APhantoms (pdu, encpdu) ∧
           ALiveIn (pdu, amaxrg, atg, ac, ala) ∧
```

$\mathtt{CC}_{\mathtt{LiveIn}}$;

$\mathtt{EnvCond} \triangleq$ *Umgebung toleriert faire Aktionen von Sys ?*

$\forall$ krq,p,kd : Enabled(SLiveMRq.fMRq(krq,p,kd)) $\Rightarrow$ (krq,p,kd) $\in$ $\mathtt{Sys.e}_{\mathtt{fMRq}}$ $\wedge$

$\forall$ p,kd : Enabled(RLiveARq.fARq(p,kd)) $\Rightarrow$ (p,kd) $\in$ $\mathtt{Sys.e}_{\mathtt{fARq}}$ $\wedge$

$\forall$ krq,d : Enabled(RLiveIn.fIn(krq,d)) $\Rightarrow$ (krq,d) $\in$ $\mathtt{Sys.e}_{\mathtt{fIn}}$ $\wedge$

$\forall$ mkrq : Enabled(MLiveIn.fMIn(mkrq)) $\Rightarrow$ mkrq $\in$ $\mathtt{Sys.e}_{\mathtt{fMIn}}$ $\wedge$

$\forall$ md : Enabled(MLiveIn.fMnoTick(md)) $\Rightarrow$ md $\in$ $\mathtt{Sys.e}_{\mathtt{fMnoTick}}$ $\wedge$

$\forall$ akrq : Enabled(ALiveIn.fAIn(akrq)) $\Rightarrow$ akrq $\in$ $\mathtt{Sys.e}_{\mathtt{fAIn}}$;

$\forall$ ad : Enabled(ALiveIn.fANoTick(ad)) $\Rightarrow$ ad $\in$ $\mathtt{Sys.e}_{\mathtt{fANoTick}}$;

IN Pars $\wedge$ Sys $\wedge$ $\Box$ EnvCond $\Rightarrow$ LiveInNoAttr *Theorem*

Anschließend beweisen wir die Gültigkeit dieses Theorems:

Satz 5.7

$$\mathtt{Pars} \wedge \mathtt{Sys} \wedge \Box\mathtt{EnvCond} \Rightarrow \mathtt{LiveInNoAttr}$$

Beweis: Der Beweis folgt der in [Lam94] vorgegebenen Methode für Liveness-Verifikationen (siehe Kapitel 3.3.3). Jede Aktion und jede Variable des SCs *LiveInNoAttr* wird im folgenden durch das Präfix LI gekennzeichnet. Wir werden nachweisen, daß die schwache Fairness der Aktion LI.fIn(krq) durch die schwache Fairness der Aktion fIn(krq,d) im Prozeß *SLiveMRq* und damit auch im System Sys impliziert wird. Dazu muß zunächst bewiesen werden, daß die Initialbedingung und die Zustandsfolgerelation von *LiveInNoAttr* durch Sys erbracht wird. Dieser Safety-Beweis wird mit Hilfe der Variablenersetzung

$$\begin{aligned} \overline{\mathtt{LI.cRq}} &= \mathtt{SLiveMRq.cRq} \wedge \\ \overline{\mathtt{LI.maxIn}} &= \mathtt{RLiveIn.maxIn} \end{aligned} \tag{5.28}$$

geführt, erfolgt nach dem im Satz 5.6 erläuterten Schema und bleibt dem Leser überlassen. Im folgenden verwenden wir das Teilergebnis dieses Beweises, daß die Aktion Sys.fIn die Aktion $\overline{\mathtt{LI.fIn}}$ impliziert.

Im Livenessbeweis wird gezeigt, daß immer, wenn $\overline{\mathtt{LI.fIn}}$ schaltbar ist, die Aktion Sys.fIn irgendwann ebenfalls schaltbar wird und ab diesem Zustand solange schaltbar bleibt, bis sie ausgeführt wird. Aufgrund der schwachen Fairness wird Sys.fIn und (wegen der oben gezeigten Implikation) auch $\overline{\mathtt{LI.fIn}}$ irgendwann geschaltet. Dadurch ist garantiert, daß LI.fIn schwache Fairness besitzt. Für diesen Beweis, verwenden wir die TLA-Regel WF2 (siehe Abbildung 3.2). Zunächst führen wir aber einige Namenskonventionen ein. SLM steht für den Bezeichner des APM *SLiveMRq*, RLA für *RLiveARq*, RLI für *RLiveIn*, RA für *RAcknowledge* und A für *Auxiliary*. MS wird für das AMC *MSDUId*, MC für *MCorruptions*, MP für *MPhantoms*, MLI für *MLiveIn*, AS für *ASDUId*, AC für *ACorruptions*, AP für *APhantoms* und ALI für *ALiveIn* eingesetzt. Im Gegensatz zu den Aktionen des Grobsystems (SC *LiveInNoAttr*), die weiterhin das Präfix LI besitzen, erhalten die Aktionen des Feinsystems kein Präfix.

$SKM + 1$ sei die Anzahl der Segmente für das Nutzdatum mit der Sequenznummer RLI.maxIn. Die Bezeichner M_i, S_T und M_T beschreiben spezielle Systemzustände von Sys:

$$M_i \triangleq i = \{k | k \in \mathtt{Nat} : k \leq SKM \wedge \exists l \in \mathtt{pkey} :: \mathtt{skn}[l] = k \wedge \mathtt{skk}[l] = \mathtt{RLI.maxIn}\} \cap \mathtt{RLI.rbk}$$

$$S_T \triangleq \{0, \ldots, SKM\}$$

$$M_T \triangleq M_{S_T}$$

`Sys` befindet sich im Zustand M_i, wenn i die Menge aller von Instanz R empfangenen Kennungen des Nutzdatums mit der Sequenznummer `RLI.maxIn` ist. Den Zustand, in dem alle Segmente des auszuliefernden Nutzdatums empfangen wurden, beschreiben wir durch den Bezeichner M_T.

Außerdem definieren wir die Invariante I_{Sys}, deren Verifikation wieder dem Leser überlassen bleibt:

$$I_{Sys} \triangleq \texttt{SLM.sba} \subseteq \texttt{RLI.rbk} \cup \{k | k \in \texttt{pkey} : \texttt{skk}[k] < \texttt{RLI.maxIn}\} \wedge \\ \exists i :: i \subseteq S_T \wedge M_i$$

I_{Sys} sagt aus, daß nur Segmentkennungen bestätigt sein dürfen, die von der Protokollinstanz R empfangen wurden, oder die Teil einer Dateneinheit sind, von der bereits ein Nachfolger an den Dienstnehmer ausgeliefert wurde.

In der TLA-Regel WF2 ersetzen wir α und β durch `fIn(krq)`, λ durch $\overline{\texttt{LI.fIn(krq)}}$, π durch die Zustandsfolgerelation $\mathcal{ZFR}$ von `Sys`, F durch M_T, T durch die Konjunktion der Livenessanforderungen `L` von `Sys` und der Invarianten I_{Sys}, w durch das Tupel `V` der Variablen von `Sys` und u durch das Tupel $\overline{\texttt{LI.V}}$ der Variablen von *LiveInNoAttr*. Damit hat WF2 die folgende Form:

$$\frac{\begin{array}{l} \langle \mathcal{ZFR} \wedge \texttt{fIn(krq,d)} \rangle_{\texttt{V}} \Rightarrow \langle \overline{\texttt{LI.fIn(krq)}} \rangle_{\overline{\texttt{LI.V}}} \\ M_T \wedge M_T' \wedge \langle \mathcal{ZFR} \wedge \texttt{fIn(krq,d)} \rangle_{\texttt{V}} \wedge \overline{\mathit{Enabled}\langle \texttt{LI.fIn(krq)} \rangle_{\texttt{LI.V}}} \\ \qquad \Rightarrow \texttt{fIn(krq,d)} \\ M_T \wedge \overline{\mathit{Enabled}\langle \texttt{LI.fIn(krq)} \rangle_{\texttt{LI.V}}} \Rightarrow \mathit{Enabled}\langle \texttt{fIn(krq,d)} \rangle_{\texttt{V}} \\ \Box[\mathcal{ZFR} \wedge \neg \texttt{fIn(krq,d)}]_{\texttt{V}} \wedge WF_{\texttt{V}}(\texttt{fIn(krq,d)}) \wedge \Box(\texttt{L} \wedge I_{Sys}) \\ \qquad \wedge \Diamond\Box \overline{\mathit{Enabled}\langle \texttt{LI.fIn(krq)} \rangle_{\texttt{LI.V}}} \Rightarrow \Diamond\Box M_T \end{array}}{\Box[\mathcal{ZFR}]_{\texttt{V}} \wedge WF_{\texttt{V}}(\texttt{fIn(krq,d)}) \wedge \Box(\texttt{L} \wedge I_{Sys}) \Rightarrow \overline{WF_{\texttt{LI.V}}(\texttt{LI.fIn(krq)})}} \tag{5.29}$$

Die erste Prämisse von Formel 5.29 wurde bereits in dem hier nicht dargestellten Safetybeweis bewiesen. Der Beweis der zweiten und der dritten Prämisse ist trivial. Im folgenden verifizieren wir die vierte Prämisse. Dabei kürzen wir die Vorbedingung der in dieser Prämisse aufgeführten Implikation durch VBR ab.

Der Beweis $VBR \Rightarrow \Diamond\Box M_T$ kann in zwei Teilbeweise zerlegt werden. Zum einen muß verifiziert werden, daß unter Einhaltung der Bedingung VBR der Zustand M_T irgendwann erreicht wird:

$$VBR \Rightarrow \Diamond M_T \tag{5.30}$$

Zum anderen muß bewiesen werden, daß M_T stabil ist:

$$VBR \wedge M_T \Rightarrow \Box M_T \tag{5.31}$$

Dieser Stabilitätsbeweis wird analog zu einem Invariantenbeweis geführt (TLA-Regel INV1) und bleibt dem Leser überlassen.

Die Formel 5.30 wird mit Hilfe der TLA-Regel LATTICE verifiziert. Dazu muß man eine Menge S angeben, über deren Elemente eine wohlfundierte Halbordnung $\succ$ definiert ist. Die Regel sagt aus, daß die Formel $\exists k \in S : P_k \leadsto U$ gilt, wenn eine Menge von Zuständen P_i mit $i \in S$ existiert und für jeden Zustand P_i eine der Formeln $\exists j \in S : (P_i \leadsto P_j) \wedge (i \succ j)$ oder $P_i \leadsto U$ nachgewiesen werden kann. Damit kann man einen temporalen Beweis in mehrere Einzelbeweise $P_i \leadsto P_j$ zerlegen, die unter Umständen nur jeweils einem Zustandsübergang

entsprechen. Als Prädikate verwenden wir M_i mit $i \in 2^{S_T}$. Die Halbordnung wird durch $i \succ j \triangleq i \subseteq j$ definiert. Sie ist wohlfundiert, da die Menge S_T das kleinste Element in der Potenzmenge S ist. Wir ersetzen in LATTICE den Bezeichner S durch 2^{S_T}, T durch VBR, U durch M_T, H_I durch M_i und die Halbordnung $i \succ j$ durch $i \subseteq j$. Die Regel LATTICE hat dann die folgende Form:

$$\frac{(VBR \wedge c \subseteq S_T) \Rightarrow (M_c \leadsto (M_T \vee \exists d :: (c \subseteq d) \wedge M_d))}{VBR \Rightarrow (\exists c :: c \subseteq M_T \wedge (M_c \leadsto M_T))} \tag{5.32}$$

Da die in der Vorbedingung VBR enthaltene Invariante I_{Sys} die Bedingung $\exists c :: c \subseteq S_T \wedge M_c$ impliziert, folgt aus der Konklusion der Regel 5.32 sofort die Formel 5.30.

Wir haben durch Anwendung des LATTICE den Beweis, daß unter der Annahme der Vorbedingung VBR der Zustand M_T irgendwann erreicht wird, in kleinere Lemmata

$$VBR \Rightarrow (M_c \leadsto \exists d :: M_d \wedge c \subseteq d) \tag{5.33}$$

zerlegt, deren Gültigkeit wir als nächstes zeigen werden. Ein Schritt in Formel 5.33 entspricht dem Empfang eines neuen Datensegments der Dateneinheit mit der Sequenznummer `RLI.maxIn` durch die Empfängerstation R. Bevor wir diese Formel verifizieren, definieren und beweisen wir in zwei Lemmata, daß eine PDU p mit der Eigenschaft $Cond(p)$ vom Basiskanal M oder A irgendwann der Instanz R bzw. S ausgeliefert wird, wenn regelmäßig PDUs q mit derselben Eigenschaft $Cond(q)$ gesendet werden. Für den Beweis der Lemmata führen wir zwei Vergangenheitsvariablen `MDeliv` und `ADeliv` ein, die die letzte von M bzw. A unverfälscht ausgelieferte PDU p speichern.

Lemma 5.7.1 *Eine PDU* p *mit einer durch* Cond(p) *beschriebenen Eigenschaft wird durch den Basiskanal M irgendwann der Empfangsinstanz R ausgeliefert, wenn regelmäßig PDUs* q *mit der Eigenschaft* Cond(q) *von der Sendeinstanz S gesendet werden:*

$$\begin{aligned} VBR \Rightarrow (\ (\ \exists p :: \Box\Diamond(Enabled\langle \exists kd, k, mk : \texttt{fMRq (k, p, kd, mk)}\rangle_V \wedge Cond(p)) \wedge \\ \Box\forall p, q :: (Cond(p) \wedge Cond(q)) \Rightarrow (p, q) \in \texttt{mla}) \\ \Rightarrow \Diamond\exists p : p \in \texttt{MDeliv} \wedge Cond(p) \wedge p \in \texttt{encpdu}) \end{aligned}$$

Beweis: Im folgenden bezeichnen wir mit VBM die Vorbedingung der Implikation in der oben angegebenen Formel. *TickCond* steht für die Bedingung des IF-Konstrukts der Aktion `MTick` in Prozeß `MLI` (AMC *MLiveIn*). Mit Hilfe der TLA-Regel LATTICE zerlegen wir den Beweis in mehrere Einzelschritte:

$$\begin{aligned} P_1 &\triangleq \exists k, p :: (k, p) \in \texttt{MLI.buf} \wedge k \geq \texttt{MLI.maxTick} \wedge Cond(p) \wedge p \in \texttt{encpdu} \\ P_2 &\triangleq \exists k, p :: (k, p) \in \texttt{MLI.buf} \wedge k = \texttt{MLI.maxTick} \wedge Cond(p) \wedge p \in \texttt{encpdu} \\ P_3 &\triangleq \exists k, p :: (k, p) \in \texttt{MLI.buf} \wedge k = \texttt{MLI.maxTick} \wedge Cond(p) \wedge p \in \texttt{encpdu} \\ &\quad \wedge \neg TickCond \\ P_4 &\triangleq \exists p :: p \in \texttt{MDeliv} \wedge p \in \texttt{encpdu} \end{aligned}$$

In diesen Einzelschritten beweisen wir die Gültigkeit der Zustandsübergänge $VBM \leadsto P_1$ und $\forall i \in \{1, 2, 3\} :: P_i \leadsto P_{i+1}$. Wir beginnen mit $VBM \leadsto P_1$, der dem Schalten der stark fairen Aktion `MRq` entspricht. Der Beweis, daß eine stark faire Aktion bestimmte Zustandsübergänge

garantiert, wird mit Hilfe der der TLA-Regel SF1 (siehe Abbildung 3.2) geführt. Zuvor wird die Invariante I des Prozesses `MLI` verifiziert, die aussagt, daß direkt nach Ausführung der Aktion `fMRq(mkrq,p)` die Behauptung $(\texttt{mkrq}, \texttt{p}) \in \texttt{MLI.buf} \wedge \texttt{mkrq} \geq \texttt{MLI.maxTick}$ gilt. Danach erfolgt der Beweis mit Hilfe der Regel SF1, in der F durch $True$, G durch P_1, T durch $VBM/\ I$, π durch $\mathcal{ZFR}$, α durch $\langle \exists kd, k, mk : \texttt{fMRq (k, p, kd, mk)} \rangle_V \wedge Cond(p)$ und ω durch `V` ersetzt wird. Der Beweis der drei Prämissen der Regel ist einfach und bleibt dem Leser überlassen.

Etwas komplizierter ist der Beweis $P_1 \leadsto P_2$, da dieser Zustandsübergang durch das mehrfache Schalten einer Aktion erbracht wird. Wir definieren das Prädikat

$$Q_i \triangleq \exists k, p :: (k, p) \in \texttt{MLI.buf} \wedge Cond(p) \wedge k = \texttt{MLI.maxTick} + i$$

Mit Hilfe der Regel WF1, in der wir F durch Q_i, G durch Q_{i-1}, π durch $\mathcal{ZFR}$, α durch $\langle fMIn(\texttt{RLI.maxTick}) \rangle_V$ und ω durch `V` ersetzen, können wir

$$Q_c \wedge (c \in \texttt{Nat}) \leadsto (\exists d \in \texttt{Nat} :: Q_d \wedge (c > d))$$

beweisen. Ein Zustandsübergang $Q_c \leadsto Q_{c+1}$ entspricht nämlich dem Schalten der schwach fairen Aktion `fMIn`. Diese Formel dient als Prämisse für die Regel LATTICE, so daß auch die Gültigkeit der Formel

$$\exists c \in \texttt{Nat} :: Q_c \leadsto Q_0$$

verifiziert werden kann. Da $P_1 \Rightarrow \exists c \in \texttt{Nat} :: Q_c$ und $Q_0 \leadsto P_2$ gilt, folgt daraus die Korrektheit von $P_1 \leadsto P_2$.

Der Beweis $P_2 \leadsto P_3$ kann wieder mit Hilfe der Regel SF1 geführt werden, in der F durch P_2, G durch P_3, T durch VBM, π durch $\mathcal{ZFR}$, α durch $\langle \texttt{fMNoTick(p)} \rangle_V \wedge Cond(p)$ und ω durch `V` ersetzt wird.

Schließlich bleibt noch der Beweis $P_3 \leadsto P_4$, den wir wieder mit Hilfe der TLA-Regel WF1 führen, in der wir F durch P_3, G durch P_4, π durch $\mathcal{ZFR}$, α durch $\langle fMIn(\texttt{MLI.maxTick}) \rangle_V$ und ω durch `V` ersetzen.

Aus den Schritten $VBM \leadsto P_1$ und $\forall i \in \{1, 2, 3\} :: P_i \leadsto P_{i+1}$ folgert man durch Anwendung der Regel LATTICE die Behauptung des Lemmas. □

Lemma 5.7.2 *Eine PDU* p *mit einer durch* Cond(p) *beschriebenen Eigenschaft wird durch den Basiskanal A irgendwann der Sendeinstanz S ausgeliefert, wenn regelmäßig PDUs* q *mit der Eigenschaft* Cond(q) *von der Empfangsinstanz R gesendet werden:*

$$\begin{aligned} VBR \Rightarrow (\ (\ \exists p :: \Box\Diamond(Enabled\langle \exists kd, ak : \texttt{fARq (p, kd, ak)} \rangle_V \wedge Cond(p)) \wedge \\ \Box\forall p, q :: (Cond(p) \wedge Cond(q)) \Rightarrow (p, q) \in \texttt{ala}) \\ \Rightarrow \Diamond\exists p : p \in \texttt{ADeliv} \wedge Cond(p) \wedge p \in \texttt{encpdu}) \end{aligned}$$

Der Beweis dieses Lemmas erfolgt analog zum Beweis von Lemma 5.7.1. □

Wir kommen nun zum Nachweis des Theorems in Formel 5.33, den wir wegen seines großen Umfangs nur skizzieren werden. Das Theorem garantiert, daß alle Segmente der Nutzdateneinheit mit der Sequenznummer `RLI.maxIn` übertragen werden. Aufgrund des Lemmas 5.7.1 reicht es aus zu zeigen, daß immer wieder PDUs vom Sender gesendet werden, in denen ein

noch nicht ausgeliefertes Segment der Dateneinheit enthalten ist. Diese Behauptung kann in die folgenden drei Formeln zerlegt werden:.

$$VBR \wedge M_i \Rightarrow \Diamond\Box(\ (\{k|\texttt{skk}[k] = \texttt{RLI.maxIn} \wedge \texttt{skn}[k] \in i\} \cap \texttt{RLI.rbk}) \subseteq \texttt{SLM.sba} \tag{5.34}$$

$$VBR \wedge M_i \Rightarrow \Box\Diamond(M_j \vee \forall k \in (j \setminus i) :: k \notin \texttt{SLM.sbr}) \tag{5.35}$$

$$VBR \wedge M_i \Rightarrow \Diamond\Box(\texttt{SLM.scred} \geq \texttt{RLI.maxIn}) \tag{5.36}$$

Durch die Formel 5.34 wird beschrieben, daß alle Datensegmente der Dateneinheit mit der Sequenznummer `RLI.maxIn`, die bereits von der Protokollinstanz R empfangen wurden, irgendwann in der Protokollinstanz S bestätigt sind. Die Formel 5.35 sagt aus, daß alle noch nicht von R empfangenen Segmente dieser Dateneinheit in Abständen von S neu gesendet werden dürfen ($k \notin$ `SLM.sbr`), da ihre Wiederholung von R immer wieder angefordert wird. Das gilt solange, bis die Datensegmente von R empfangen werden (M_j). Die Formel 5.36 beschreibt, daß der Sendekredit in S irgendwann so groß ist, daß Segmente der Dateneinheit mit der Sequenznummer `RLI.maxIn` gesendet werden dürfen.

Der Beweis dieser Formeln folgt aus dem Lemma 5.7.2 und aus dem APM *RLiveARq*, das sicherstellt, daß die Empfangsinstanz R regelmäßig Bestätigungen für alle Dateneinheiten mit Sequenznummern kleiner als `RLI.maxIn` und für alle empfangenen Segmente der Dateneinheit mit dieser Sequenznummer sendet (Aktion `fARq`). Außerdem sendet R regelmäßig Wiederholungsanforderungen für alle noch nicht empfangenen Segmente dieser Dateneinheit. Ferner gibt R seinem Partner in Abständen einen neuen Sendekredit, der immer die Dateneinheit mit der Sequenznummer `RLI.maxIn` einschließt.

Nun ist nach APM *RLiveARq* die Aktion `fARq` allerdings nur schaltbar, wenn zuvor eine explizite Anforderung einer Bestätigung, Fehlermeldung oder eines neuen Kredits von der Senderinstanz S eingetroffen ist. Da S nach APM *SLiveMRq* jedoch in regelmäßigen Abständen derartige Anforderungen über Kanal M absendet, treffen einige dieser Anforderungen nach Lemma 5.7.1 in R ein, so daß die Aktion `fARq` regelmäßig schaltbar ist. Dadurch treffen Bestätigungen, Wiederholungsanforderungen und Kreditvergabemitteilungen aufgrund des Lemmas 5.7.2 irgendwann in der Sendeinstanz S ein, und die Formeln 5.34, 5.35 und 5.36 sind wahr.

Die drei Formeln stellen zusammen mit dem APM *SLiveMRq* sicher, daß noch nicht empfangene Segmente der Dateneinheit mit der Sequenznummer `RLI.maxIn` regelmäßig von Instanz S über den Kanal M gesendet werden. Da sie aufgrund von Lemma 5.7.1 irgendwann von der Instanz R empfangen werden, gilt die Formel 5.33. Daraus folgt jedoch die Gültigkeit der vierten Prämisse der in Formel 5.29 aufgestellten TLA-Regel WF2, aus deren Konklusion sofort die Behauptung des Satzes 5.7 folgt. □

Wir müssen nun noch den Beweis erbringen, daß die fairen Aktionen des Systems `Sys` zu allen Safety-APMs des Frameworks mit Ausnahme des APMs *DataChanOpenR* kompatibel sind. Der oben geführte Beweis hat nur Bestand, wenn `Sys` nicht mit Prozessen gekoppelt wird, die die Schaltbedingung der fairen Aktionen einschränken. Zum Beispiel wäre nach der Kopplung von `Sys` mit einem APM, das das Schalten von Aktion `fMRq` immer ausschließt, der Satz nicht mehr beweisbar. Der Ausschluß derartiger Kopplungen wird durch die Bedingung $\Box$*EnvCond* beschrieben. In einem System, das `Sys` als Prozeß enthält, muß jede Aktion π_k, die Teil einer fairen Aktion `Sys.`π ist, schaltbar sein, wenn auch `Sys.`π selbst schaltbar ist. Die Kompatibilität von `Sys` zu allen APMs P wird nachgewiesen, indem für das aus `Sys` und

P komponierte System Sys_P

$$Sys_P \Rightarrow \Box EnvCond$$

durch einen Invariantenbeweis verifiziert wird. Dabei nehmen wir an, daß die Prozeßparameter von P immer durch dieselben Werte wie die namensgleichen Prozeßparameter in Sys ersetzt werden.

Wie schon ausgeführt wurde, ist das Theorem nicht zu dem APM *DataChanOpenR* der Intraconnection-Coordination kompatibel, das das Öffnen und Schließen der Datenkanäle beschreibt. Man kann jedoch leicht beweisen, daß dieses APM problemlos auch zu dem System `Sys` komponiert werden kann, ohne die Bedingung `EnvCond` des Theorems zu verletzen, wenn in der Dienstbeschreibung auch das äquivalente SC *DataChanOpenR* enthalten ist. Dadurch wird sichergestellt, daß die faire Aktion `fIn` von `Sys` nur in den Zuständen von APM *DataChanOpenR* blockiert ist, in denen die Aktion `LiveIn.fIn` der Dienstspezifikation durch das SC *DataChanOpenR* ebenfalls blockiert ist. Wenn derartige Zustände unendlich oft auftreten, ist die Fairness der Aktion `fIn` im SC per definitionem garantiert. Ansonsten stören die Einschränkungen der Aktion `fIn` von `Sys` den Beweis des Satzes 5.7 nicht.

Das Framework enthält drei weitere Theoreme zur Verifikation der Liveness-SCs des Datentransfers. Zwei Theoreme dienen zum Beweis des SCs *LiveIn*. Eines dieser Theoreme ist für Protokolle vorgesehen, die eine Behebung von Übertragungsfehlern anbieten. Es stellt sicher, daß eine Dateneinheit d mit einer bestimmten Eigenschaft $Cond(d)$ irgendwann dem Dienstnehmer ausgeliefert wird, wenn genügend häufig Dateneinheiten mit dieser Eigenschaft gesendet werden. Das Theorem verwendet das gleiche Subsystem wie das oben bewiesene Theorem von SC *LiveInNoAttr*. Allerdings ist die Parameterbedingung `Pars` stärker, da der Basiskanal M die Auslieferung einer PDU garantieren muß, die eine Dateneinheit mit $Cond(d)$ enthält, wenn diese häufig genug gesendet wird.

Das zweite Theorem wird dagegen für Protokolle verwendet, die keine Fehlerbehebung durch Übertragungswiederholung verwenden. Es garantiert ebenfalls die Auslieferung einer Dateneinheit d mit der Eigenschaft $Cond(d)$, wenn regelmäßig Daten mit dieser Eigenschaft gesendet werden. Das Subsystem `Sys` setzt sich aus den APMs *SLiveMRq* und *RLiveIn* sowie den AMCs *MSDUId*, *MCorruptions*, *MPhantoms* und *MLiveIn* zusammen.

Das vierte Theorem des Frameworks dient zur Verifikation des SCs *LiveCf*. Es berücksichtigt den Mechanismus, daß alle ausgelieferten Dateneinheiten von Instanz R bestätigt werden. Außerdem sichert die Fairness der Aktion `fCf` im APM *SLiveCf* zu, daß alle bestätigten Dateneinheiten dem Dienstnehmer an dem Dienstzugangspunkt zu Instanz S gemeldet werden. Das Theorem verwendet ein System `Sys`, das aus den APMs *SLiveMRq*, *SLiveCf*, *RLiveARq* und *RAcknowledge* sowie den AMCs *MSDUId*, *MCorruptions*, *MPhantoms*, *MLiveIn*, *ASDUId*, *ACorruptions*, *APhantoms* und *ALiveIn* komponiert ist.

5.6.3 Theoreme zu den SCs der Intraconnection-Coordination

Wie wir schon bei der Vorstellung der APMs der Intraconnection-Coordination erläuterten (siehe Abschnitt 5.3.3), haben die meisten dieser APMs dasselbe Aussehen wie die durch sie realisierten SCs. Die Theoreme zum Beweis der SCs sind deshalb trivial. Bei den APMs und SCs dieser Gruppe handelt es sich um *ConnectionState*, *DataChanOpenW*, *DataChanOpenR*, *StopOld*, *UserBehaviour* und *UserHSReaction*.

Das einzige Safety-SC, zu dessen Implementierung mehrere APMs verwendet wurden, ist *DelayedClose*. Es wird durch ein Subsystem `Sys` impliziert, das sich aus den APMs *SBuf-*

ferKey, *SBufferUsd*, *SMarkFinal*, *RBufferKey*, *RBufferUsd*, *RMarkFinal*, *RDeliverBuff* und *DataChanOpenW* sowie den AMCs *MSDUId*, *MCorruptions* und *MPhantoms* zusammensetzt. Die AMCs garantieren, daß die Sequenznummer der zuletzt gesendeten Dateneinheit unverfälscht über den Basiskanal M übertragen wird.

Das System `Sys`, das das Liveness-SC *LivePAbI* implementiert, besteht aus den APMs *SLivePAbI*, *SAcknowledge* und *RAcknowledge* und den AMCs *ASDUId*, *ACorruptions* und *APhantoms*. Das APM *SLivePAbI* stellt sicher, daß ein Verbindungsabbruch durch die Protokollinstanz S initiiert wird, wenn mindestens eine Dateneinheit dauerhaft unbestätigt bleibt. Da alle von Instanz R nicht auslieferbaren Dateneinheiten nicht bestätigt werden können (APM *RAcknowledge*), garantiert die schwache Fairness der Aktion `fPAbI` im APM *SLivePAbI* die schwache Fairness der Aktion `fPAbI` im SC *LivePAbI*.

5.7 Theoreme zu den APMs

In Abschnitt 5.7.1 führen wir die Theoreme zum Beweis der APMs des Datentransfers und der Intraconnection-Coordination ein. Die Theoreme zum Beweis der Interconnection-Coordination werden in Abschnitt 5.7.2 vorgestellt.

5.7.1 Theoreme zu den APMs des Datentransfers und der Intraconnection-Coordination

Aus den APMs konnten wir FAPMs herleiten, da sich in den APMs die Sequenznummern aller aktiven Dateneinheiten in einem Intervall mit fester Länge I_R befinden (siehe Abschnitt 5.5.1). Wir konnten deshalb die endliche Menge $\texttt{fkey} \triangleq \{0, \ldots, I_R - 1\}$ einführen. Für die aktiven Dateneinheiten wurden die Sequenznummern N_{nat} aus der Menge `Nat` nach der Formel 5.15 durch Sequenznummern N_{mod} aus `fkey` ersetzt, so daß die FAPMs nur eine endliche Menge an Sequenznummern speichern müssen.

Um Theoreme zum Beweis der APMs angeben zu können, muß eine surjektive Abbildung von den FAPM-Zuständen zu den APM-Zuständen gefunden werden, für die die Refinement Mapping-Eigenschaften (siehe Abschnitt 3.3.4) nachgewiesen werden kann. Aufgrund der endlichen Definitionsmenge und der unendlichen Wertemenge kann jedoch keine Abbildung surjektiv sein. In den FAPMs des Datentransfers werden zum Beispiel die veralteten Sequenznummern, die das Systemverhalten nicht mehr beeinflussen können, aus den Puffern entfernt, während sie in den APMs weiterhin gespeichert bleiben und deshalb auf den Systemzustand abgebildet werden. Dieses Problem lösen wir durch Speicherung der Sequenznummern inaktiv gewordener Dateneinheiten in Vergangenheitsvariablen, mit deren Hilfe man eine surjektive Abbildung finden kann, die Refinement Mapping-Eigenschaften besitzt. Durch die Vergangenheitsvariablen wird allerdings der Zustandsraum der FAPMs unendlich. Da sie jedoch nicht das Systemverhalten beeinflussen und deshalb nicht implementiert werden müssen, bleibt die Anzahl der Zustände implementierter Variablen weiterhin endlich.

Bei der Erstellung der Theoreme berücksichtigen wir die in Abschnitt 5.5.1 verifizierte Eigenschaft, daß man die Sequenznummern N_{mod} aus der Menge `fkey` gemäß den Formeln 5.16 (Instanz S) und 5.17 (Instanz R) in Sequenznummern N_{nat} überführen kann. Dazu muß die Systemspezifikation `Sys` die FAPMs *SBufferKey*, *RBufferKey*, *SCapacity*, *SAcknowledge*, *RAcknowledge* und *SSendInOrder* sowie die AMCs *MSDUId*, *ASDUId*, *MCorruptions*, *ACorruptions*, *MReorderings*, *MCapacity* und *AReorderingsMRq* enthalten. Mit Hilfe dieser

Formeln und der Vergangenheitsvariablen können wir Refinement Mappings zwischen den Variablen der FAPMs und denjenigen der APMs angeben. Dabei werden die Sequenznummern der aktiven Dateneinheiten mit Hilfe der Formeln 5.16 bzw. 5.17 aus den Variablen der FAPMs abgebildet. Die Sequenznummern der inaktiv gewordenen Dateneinheiten entsprechen hingegen den Werten der Vergangenheitsvariablen.

Die Formeln 5.16 und 5.17 verwenden Zähler `cRq` bzw. `maxIn` aus der Menge `Nat`, die in den FAPMs nicht definiert sind. Wir ersetzen sie durch zwei Vergangenheitsvariablen `HcRq` und `HmaxIn`, die in den folgenden FAPMs *SHcRq* und *RHmaxIn* definiert sind:

```
PROCESS SHcRq      Fuehrt Vergangenheitsvariable HcRq ein
IMPORT Constants;
BODY
  VARIABLES
    HcRq : key;        Hilfssequenznummernzaehler
  INIT ≙ HcRq = 0 ;
  ACTIONS
    Rq ≙ Uebergabe vom Dienstnehmer
      HcRq' = HcRq + 1;
END

PROCESS RHmaxIn   Fuehrt Vergangenheitsvariable HmaxIn ein
                ( sc, mb, rcu, rcc, c, tr, ir : Nat )
IMPORT Constants;
BODY
  VARIABLES
    HmaxIn : key; Hilfszaehler fuer die hoechste Sequenznummer der ausgelieferten Daten
  INIT ≙ HmaxIn = 0 ;
  ACTIONS
    In ( krq : fkey ) ≙   Auslieferung an den Dienstnehmer
      HmaxIn' = IF (krq = "notsent")
                THEN HmaxIn
                ELSE max(HmaxIn, (krq - HmaxIn + sc + rcu + rcc +
                         mb · (c + tr)) mod ir +
                         HmaxIn - sc - rcu - rcc - mb · (c + tr));
END
```

Als Beispiel zeigen wir nun das Theorem zum Beweis des APMs *SBufferKey*. Dazu benötigen wir ein FAPM *SHBufferKey*, das die Vergangenheitsvariable `Hsbk` der Variablen `sbk` enthält. (Die Vergangenheitsvariable des Zählers `cRq` ist `HcRq` im FAPM *SHcRq*.) Wir können damit das folgende Theorem aufstellen:

```
LET Pars ≙ Anforderungen an die Prozesstypparameter
           mtc ⊆ { (p, q) | q ∉ encpdu ∨ skey[spci[p] ] = skey[spci[q] ] } ∧
           atc ⊆ { (p, q) | q ∉ encpdu ∨ sack[spci[p] ] = sack[spci[q] ]} ∧
           ir ≥ 2 · sc + 2 · rcu + 2 · rcc + mb · (c + tr) + 1;
   Sys ≙   Subsystem aus APMs und AMCs, durch das SBufferKey erfuellt wird
           SBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn, skm,
                       usdsize, mb, ir) ∧
           SAcknowledge (pdu, pci, encpdu, spci, sack, ir) ∧
           SCapacity (pdu, pci, usd, encpdu, spci, sack, skk, skn, skm,
                      sc, ir) ∧
           RBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn, skm,
                       usdsize, rcu, rcc, ir) ∧
```

```
            RAcknowledge (pdu, pci, encpdu, spci, sack, skk, rcu, rcc, ma, ir) ∧
            MSDUId ∧
            MCorruptions (pdu, mtc) ∧
            MPhantoms (pdu,encpdu) ∧
            MReorderings (tr) ∧
            MCapacity (c) ∧
            ASDUId ∧
            ACorruptions (pdu, atc) ∧
            APhantoms (pdu, encpdu) ∧
            AReorderingsMRq (tr) ∧
            SHcRq ∧
            SHVariables ( sc, mb, rcu, rcc, c, tr, ir ) ∧
            RHmaxIn ( sc, mb, rcu, rcc, c, tr, ir ) ∧
            CC_SBufferKey;
IN Pars ∧ Sys ⇒ SBufferKey (pdu, pci, usd, encpdu, spci, skey, skk, skn,
                            skm, usdsize, mb)    Theorem
```

Die FAPMs *SHcRq*, *SHVariables* und *RHmaxIn*, die nur Vergangenheitsvariablen enthalten, werden in der Theorembeschreibung im Framework nicht explizit aufgeschrieben, da sie nur zur Verifikation dienen, für den Protokollentwickler jedoch nicht relevant sind. Die anderen Theoreme zum Beweis der APMs des Datentransfers und der Intraconnection-Coordination haben eine zu diesem Theorem analoge Form.

5.7.2 Theoreme zu den APMs der Interconnection-Coordination

Bei der Entwicklung der FAPMs dieser Gruppe wurde ein neuer Protokollmechanismus modelliert, der eine endliche Anzahl an Verbindungskennungen verwaltet (siehe Abschnitt 5.5.2). Einer neu aufgebauten Verbindung wird eine Kennung zugeordnet, die sie eindeutig identifiziert. Erst nach dem endgültigen Abbau der Verbindung kann ihre Kennung einer anderen Verbindung zugewiesen werden.

Das Verfahren wird durch die FAPMs *AssignCNo*, *SelectCNo*, *FreeCNo* und *LiveFreeCNo* modelliert, die zusammen mit dem FAPM *OpenConnections* zum Beweis des APMs *OpenConnections* herangezogen werden. In der Variablen `opcon` des APMs ist die Menge der an einem Rechnernetzknoten aktiven Verbindungen gespeichert. Das Refinement Mapping für die Variable `opcon` wird unter Verwendung von zwei Vergangenheitsvariablen erbracht. Eine Vergangenheitsvariable speichert die Menge aller Verbindungen, denen vom aktuellen Knoten schon eine Kennung zugeordnet wurde. Die andere Variable speichert die Verbindungen, deren Kennungen bereits wieder freigegeben wurden. `opcon` entspricht der Differenz dieser beiden Mengenvariablen.

FindDestination, das zweite APM der Interconnection-Coordination, entspricht dem gleichnamigen FAPM, so daß man ein triviales Theorem zu seinem Beweis angeben kann.

Kapitel 6

Eine Anwendung: XTP 4.0

Das Hochleistungstransportprotokoll XTP ((e)Xpress Transfer Protocol; vgl. [Pro89], siehe auch Abschnitt 2.4.4) eignet sich gut als Fallbeispiel für den Einsatz des Frameworks. Zum einen scheint sich XTP als Standardtransportprotokoll für schnelle Netze durchzusetzen. Ein XTP-Derivat wurde zum Beispiel als Transportkomponente für den asynchronen Transfermode ATM vorgeschlagen (vgl. [dP94]). Zum anderen enthält XTP alle wichtigen Architekturmerkmale von Hochleistungsprotokollen (siehe Abschnitt 2.3). Die einzelnen Protokollfunktionen (siehe Abschnitt 2.3.3) wurden als eigenständige, voneinander im wesentlichen unabhängig ablaufende Einheiten definiert und können je nach Anforderung des Transportdienstnehmers zu- oder abgeschaltet werden. Dadurch ist eine dynamische anwendungsabhängige Konfiguration des Protokolls möglich (siehe Abschnitt 2.3.5). Die aktuelle Version 4.0 [XTP95] ermöglicht sogar eine dynamische Rekonfigurierbarkeit des Protokolls innerhalb von Verbindungen. Zudem ist XTP von seinen Entwicklern für Implementierungen in Hardware und für die Parallelisierung seiner Funktionen vorgesehen[1].

Zur Erläuterung unserer Vorgehensweise bei der formalen Spezifikation und Verifikation von XTP mit dem Framework werfen wir zunächst einen Blick auf die Beschreibung der Version 4.0 von XTP [XTP95]. Wie in vielen vergleichbaren Protokollbeschreibungen wird zwar das Format der Protokolldateneinheiten (PDUs) von XTP sehr ausführlich erläutert. Das Ablaufverhalten der Protokollinstanzen, das auch zu einer Protokollspezifikation gehört, ist hingegen im wesentlichen informell und nicht immer eindeutig. Formale Mittel (ausschließlich Zustandsübergangsdiagramme) werden lediglich zur Beschreibung der Zustände beim Verbindungsaufbau und -abbau verwendet. Einige Aspekte wie die Behandlung schwerer Protokollfehler werden nur oberflächlich behandelt. Das größte Manko ist jedoch das Fehlen einer Transportdienstbeschreibung. Weder die Dienstprimitive noch ihr Ablaufverhalten sind in [XTP95] beschrieben. Deshalb müssen wir auf der Basis der XTP-Protokollbeschreibung zunächst eine eigene Transportdienstspezifikation entwickeln. Dazu werden SCs des Frameworks parametrisiert und miteinander zu einer Dienstspezifikation komponiert. Anschließend erzeugen wir Spezifikationen der von XTP verwendeten Protokollmechanismen durch Parametrisierung von FAPMs aus dem Framework. Die für die Ersetzung der Parameter notwendigen Daten sind in [XTP95] bis auf wenige Ausnahmen ausführlich beschrieben, so daß geeignete Parameterbelegungen der FAPMs einfach ermittelt werden können. Die Kompo-

[1] Ein weiterer praktischer Grund für unsere Auswahl besteht darin, daß die XTP-Beschreibung im Gegensatz zu Beschreibungen einiger anderer Protokolle frei verfügbar ist.

sition der FAPMs bildet eine formale Spezifikation von XTP. Durch Parametrisierung von APMs erzeugen wir auf die gleiche Weise auch eine abstrakte Protokollspezifikation von XTP. In weiteren Schritten weisen wir unter Verwendung der Theoreme des Frameworks nach, daß die Dienstspezifikation von der abstrakten Protokollspezifikation und die abstrakte Protokollspezifikation von der XTP-Protokollspezifikation implementiert wird.

Weiterhin überprüfen wir, ob unsere XTP-Spezifikation zu der in [XTP95] vorgestellten informellen XTP-Beschreibung konform ist. Bei den Verbindungen zwischen zwei Dienstnehmern[2] wurden einige Widersprüche in der XTP-Beschreibung gefunden, die auch vom XTP-Forum, dem Gremium der Protokollentwickler, anerkannt wurden. Zuletzt müssen wir untersuchen, ob unsere Transportdienstspezifikation die von Hochleistungsanwendungen geforderten Diensteigenschaften (siehe Abschnitt 2.3.1) erfüllt. Dadurch wird sichergestellt, daß XTP sich unter funktionalen Gesichtspunkten tatsächlich als Hochleistungstransportprotokoll eignet. Die Vorgehensweise bei der Spezifikation und Verifikation von XTP sowie die Ergebnisse wurden bereits in [HK97a] vorveröffentlicht.

Nach einer Übersicht über die Funktionalität von XTP in Abschnitt 6.1 beschreiben wir die Instantiierung der Prozeßtypen des Frameworks. Da wir die SCs aufgrund des Fehlens einer Dienstbeschreibung auf der Basis der XTP-Protokollmechanismen und der durch sie erbrachten Dienstleistungen entworfen haben, werden wir zuerst in Abschnitt 6.2 die Entwicklung der XTP-Protokollspezifikationen erörtern. Dadurch erhält der Leser einen Einblick in die Funktionalität der Protokollmechanismen, wodurch das Verständnis des Entwurfs der SCs, der in Abschnitt 6.3 beschrieben ist, erleichtert wird. In Abschnitt 6.4 zeigen wir die Verifikation, daß zum einen die APMs durch die FAPMs und zum anderen die SCs durch die APMs erfüllt werden. Die Bewertung der funktionalen Eigenschaften von XTP erfolgt in Abschnitt 6.5. Wir schließen das Kapitel mit einigen Überlegungen zum Einsatz des kompositionalen Stils in der Praxis (Abschnitt 6.6).

6.1 Übersicht

Wie in Abschnitt 2.4.4 schon erwähnt wurde, zeichnet sich XTP vor allem dadurch aus, daß die XTP-Protokollfunktionen abhängig von der erforderlichen Dienstgüte durch den Dienstnehmer ein- und abgeschaltet werden können. Wir skizzieren im folgenden die XTP-Protokollfunktionen für den Datentransfer (Abschnitt 6.1.1) sowie für die Verbindungsverwaltung (Abschnitt 6.1.2). Danach wird in Abschnitt 6.1.3 das Format der XTP-Protokolldateneinheiten (PDUs) beschrieben.

6.1.1 Protokollfunktionen des Datentransfers

XTP stellt dem Dienstnehmer eine Reihe von Übertragungsmodi zur Verfügung, die sich voneinander durch unterschiedliche Funktionen der Fehlerbehandlung und der Flußkontrolle unterscheiden (siehe Abschnitt 2.2.2). Die Modi werden dem Verbindungspartner durch spezielle Steuerbits in den PDUs angezeigt.

Die Erkennung von Bitfehlern in PDUs erfolgt durch Prüfsummen mit einer Länge von zwei Bytes. Der Dienstnehmer kann dabei auswählen, ob die Prüfsummmen nur Paketköpfe

[2]Da das Framework noch keine Mehrbenutzerverbindungen unterstützt, wurde auf die Untersuchung des XTP-Multicasts verzichtet.

oder ganze Pakete sichern. Für die Sicherung gegen Reihenfolgevertauschungen und Verluste wird jedem Nutzdatenbyte eine eigene acht Byte lange Sequenznummer zugeordnet. Für die Meldung von Übertragungsfehlern steht ein PDU-Typ zur Verfügung, in dem der Sendeinstanz die Sequenznummer des ersten Nutzdatenbyte, das noch nicht korrekt empfangen wurde, übermittelt wird. Außerdem werden dem Sender auch zusammenhängende Datenbereiche (Spans) mitgeteilt, die korrekt empfangen wurden, obwohl vorhergehende Nutzdaten fehlerhaft waren. Durch die Art der Verwendung von Spans kann sowohl Übertragungswiederholung durch Rücksetzung als auch durch selektive Wiederholung realisiert werden. Die Behandlung von Übertragungsfehlern am Ende eines Datenstroms ist in [XTP95] allerdings so unvollständig und widersprüchlich beschrieben, daß sie zu fehlerhaften Protokollimplementierungen führen kann. In Abschnitt 6.5 gehen wir näher auf diese Problematik ein. Zwei unterschiedliche Verfahren zur Initiierung von Fehlerbehebungsmaßnahmen stehen zur Verfügung. Im ersten Verfahren kann der Sender von Daten seinem Partner eine Anforderung übermitteln, in der eine Bestätigung der korrekt empfangenen Dateneinheiten angefordert wird. Aus dieser Bestätigung kann er die fehlerhaften Daten ermitteln und wiederholen. Dieser Mechanismus entspricht der Methode PAR (Positive Acknowledgement with Retransmission; siehe Abschnitt 2.2.2). Im zweiten Verfahren kann neben dem Sender auch der Empfänger von sich aus eine Bestätigung absetzen, wenn er einen Übertragungsfehler erkannt hat. Somit unterstützt XTP auch die Methode ARQ (Automatic Repeat Request). Außerdem kann die Behebung von Fehlern vollständig abgeschaltet werden.

Die Flußkontrolle erfolgt sowohl durch Kreditvergabeschemata als auch durch temporäre Sendebeschränkungen. Im Kreditvergabemechanismus sendet der Empfänger von Dateneinheiten seinem Partner als Information die Sequenznummer, bis zu der Dateneinheiten gesendet werden dürfen. Zusammen mit der Sequenznummer der Dateneinheit, die als erste noch nicht korrekt empfangen wurde, wird dadurch ein Sendefenster definiert, in dem alle gesendeten Dateneinheiten liegen müssen. Dieser Fenstermechanismus kann abgeschaltet werden, so daß der Sender nicht mehr an das Sendefenster gebunden ist. Weiterhin kann festgelegt werden, daß der Empfänger nur die Sendung von Dateneinheiten erlaubt, für die er Empfangsspeicherplatz garantieren kann (Reservation Mode), oder daß er die Sendung zusätzlicher Dateneinheiten erlaubt, die der Empfänger unter Umständen verwerfen muß. Bei der Flußkontrolle durch temporäre Sendebeschränkungen wird die Methode der auf Datenraten basierenden Sendesteuerung verwendet. Das heißt, daß die Sendung einer bestimmten Dateneinheit unter Umständen für eine bestimmte Zeit verzögert werden muß, wenn sonst die erlaubte Datenrate überschritten würde. Da diese Art der Flußkontrolle für die Liveness von XTP nicht relevant ist (siehe Abschnitt 5.3.1) und das Framework im aktuellen Zustand noch keine Realzeitmechanismen kennt, haben wir auf eine Spezifikation dieses Protokollmechanismus verzichtet.

6.1.2 Protokollfunktionen der Verbindungsverwaltung

Auch für das Management von Verbindungen unterstützt XTP unterschiedliche Protokollfunktionen (siehe auch Abschnitt 2.2.3). Der Verbindungsaufbau erfolgt immer implizit. Der Initiator einer Verbindung darf unmittelbar nach Sendung der Verbindungsaufbau-PDU, die selbst Nutzdaten enthalten darf, mit der Sendung von Daten-PDUs beginnen. Er legt fest, ob er einen Simplex-Datenkanal, auf dem nur er senden darf, oder einen Duplex-Datenkanal aufbauen möchte. Sein Partner darf nach dem Erhalt der Verbindungsaufbau-PDU mit der

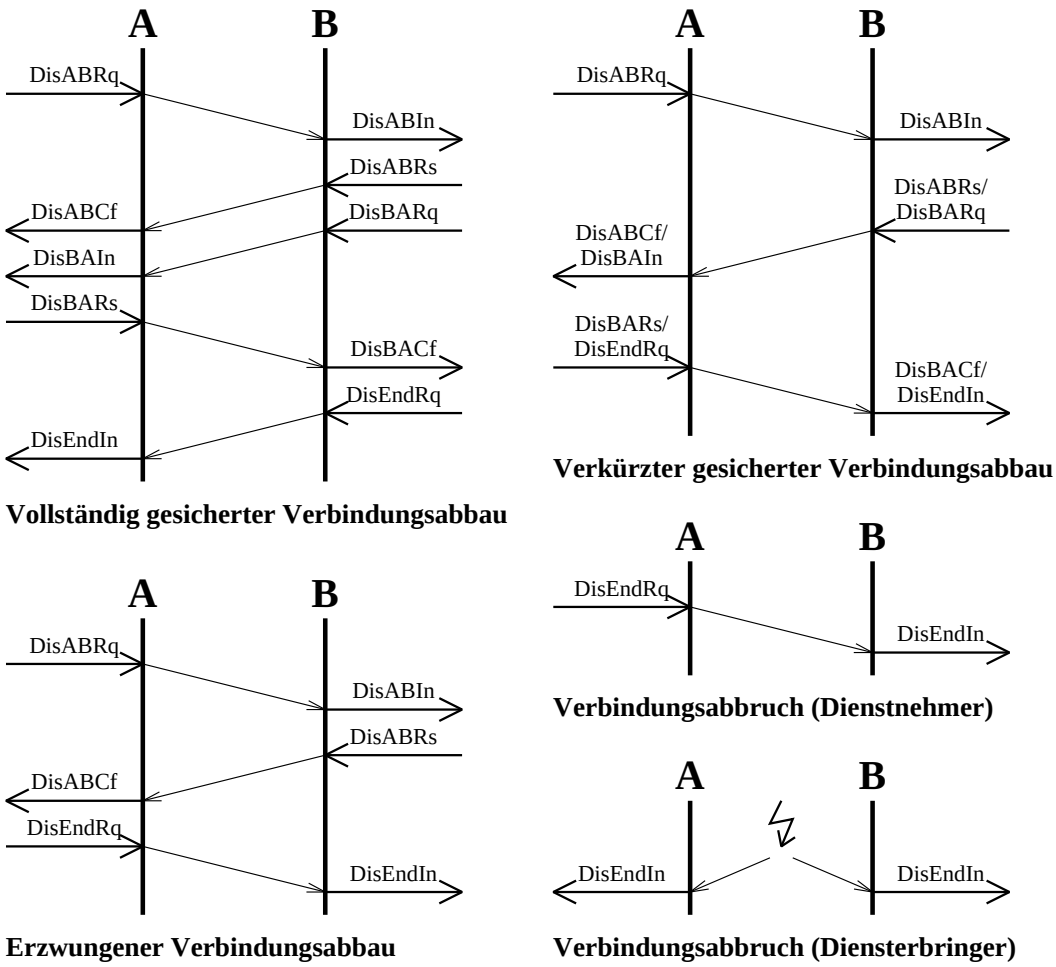

Abbildung 6.1: XTP-Verbindungsabbauverfahren für Duplex-Datenkanäle

Sendung von Daten anfangen, wenn ein Duplex-Datenkanal aufgebaut wird.

XTP sieht verschiedenartige Verbindungsabbauverfahren vor (siehe Abbildung 6.1). Beim Verbindungsabbau wird ein Duplex-Datenkanal immer als ein Paar gegenläufiger, voneinander unabhängiger Simplex-Datenkanäle aufgefaßt, die separat abgebaut werden. Beim vollständig gesicherten Verbindungsabbau (links oben in Abbildung 6.1) werden zunächst die Simplex-Kanäle durch bestätigte Verbindungsabbaudienste *DisAB* bzw. *DisBA* abgebaut. Erst danach wird die Verbindung endgültig aufgegeben (Dienst *DisEnd*). Beim verkürzten Verbindungsabbau (rechts oben) wird die Bestätigung des Verbindungsabbaus *DisAB* mit dem Verbindungsabbau *DisBA* und die Bestätigung von *DisBA* mit dem Verbindungsende *DisEnd* kombiniert, um den Verbindungsabbauprozeß zu verkürzen. Außerdem unterstützt XTP den erzwungenen Verbindungsabbau (links unten), bei dem Station *A* nach Bestätigung des Verbindungsabbaus *DisAB* ohne Rücksicht auf ihren Partner terminiert, einen unbestätigten Verbindungsabbruch (rechts Mitte) sowie einen vom Diensterbringer initiierten Verbindungsabbruch (rechts unten), der aufgrund eines Protokollfehlers oder andauernder Passivität der

Datenkanäle erfolgen kann. Der Verbindungsabbau wird durch drei Flags in den PDUs gesteuert, mit denen eine Station A ihrem Partner B die Schließung des Datenkanals *DisAB*, die Bestätigung der Schließung des Kanals *DisBA* und das Verbindungsende *DisEnd* anzeigen kann. Wurde beim Verbindungsaufbau ein Simplex-Kanal (z.B. von B nach A) angefordert, muß bei allen Verbindungsabbauverfahren selbstverständlich nur ein Datenkanal abgebaut werden (*DisBA*).

Datagramme, bestätigte Datagramme und Transaktionen werden durch kurzzeitige Verbindungen realisiert. Ein unbestätigtes Datagramm wird durch eine PDU übertragen, die zugleich den Aufbau eines Simplex-Datenkanals anfordert, Nutzdaten überträgt und einen unbestätigten Verbindungsabbruch anzeigt. Bei bestätigten Datagrammen wird anstatt des Verbindungsabbruchs ein verkürzter gesicherter Verbindungsabbau angezeigt, so daß der Partner in seiner Bestätigung die Verbindung abbaut. In einer Transaktion wird der vollständige gesicherte Verbindungsabbau verwendet. In einer fehlerfreien Verbindung werden somit genau drei PDUs übertragen (vgl. [SW90]).

Ein weiteres wichtiges Element der Verbindungsverwaltung ist die Verhandlung neuer Datenraten (Traffic Specification Negotiation). Damit wird zum einen die auf Datenraten basierende Sendesteuerung an die aktuelle Netzbelastung angepaßt. Zum anderen kann man mit dieser Methode auch den Datentransfermodus umschalten. Der Initiator der Verhandlung sendet eine PDU, die für beide Senderichtungen seine Wünsche bzgl. der mittleren Datenrate und der Datenrate bei Bursts enthält. Der Empfänger bestätigt diese Anforderungen, wobei er aber auch niedrigere Datenraten angeben kann. Wird der Datentransfermodus während der Verhandlung umgeschaltet, darf der Initiator zwischen Sendung und Bestätigung der Aushandlung keine neuen Dateneinheiten senden, da diese weder dem alten noch dem neuen Modus zugeordnet werden können[3]. Der Empfänger darf die Verhandlung nur bestätigen, wenn entweder alle Dateneinheiten fehlerfrei ausgeliefert wurden, oder wenn die Fehlerbehebung im alten Modus ausgeschaltet ist. Damit der Initiator einer Verhandlung eine empfangene Bestätigung zu der entsprechenden Anforderung zuordnen kann, wird den PDUs jeder Verhandlung eine eigene Kennung zugeordnet.

Die Verhandlung neuer Datenraten wird auch zur Identifikation von Verbindungskontexten verwendet. Um nicht in allen PDUs die vollständige Adreßinformation der beteiligten Partner mitführen zu müssen, wird diese nur in der Verbindungsaufbau-PDU übertragen[4]. Die Kontext-Zuordnung erfolgt durch einen 63 Bit langen Schlüssel, der vom Initiator der Verbindung zugeteilt wird. Der Initiator hat dafür Sorge zu tragen, daß dieser Schlüssel lokal eindeutig ist und PDUs aus alten Verbindungen, die denselben Schlüssel verwendeten, nicht mehr empfangen werden können. Durch ein zusätzliches Return-Flag wird angezeigt, ob der Schlüssel einer PDU von deren Sender oder Empfänger generiert wurde. Für die Zuordnung einer empfangenen PDU zu einem Verbindungskontext ist es von Vorteil, wenn der in der PDU enthaltene Schlüssel in der Empfängerstation eindeutig ist. Um auch der Station eine schnelle Kontexterkennung zu ermöglichen, die die Verbindung nicht aufgebaut hat, kann sie bei Verhandlungen von Datenraten einen selbst generierten Schlüssel übertragen, den ihr Partner von da ab in allen gesendeten Paketen verwendet.

[3] XTP verbietet Datenübertragungen während der Modusumschaltung zwar nicht explizit, empfiehlt Implementierern jedoch ausdrücklich, sie in Realisierungen nicht vorzusehen.

[4] Außerdem werden vollständige Adressen angegeben, wenn in einer Multicast-Verbindung eine neue Station zugeschaltet wird.

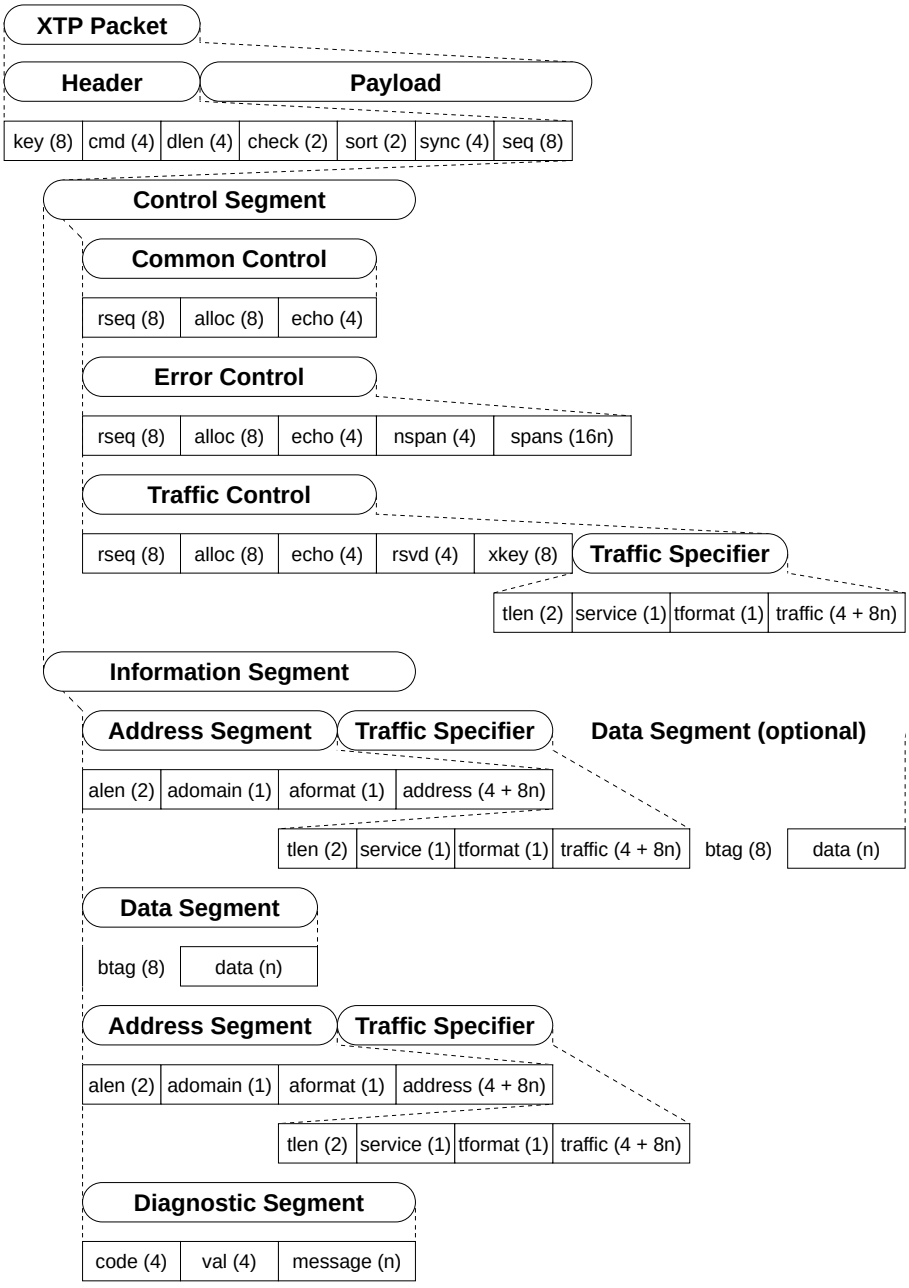

Abbildung 6.2: XTP-PDU-Format (entnommen aus [XTP95])

XTP kennt sieben verschiedene PDU-Formate, die alle einen gemeinsamen 32 Byte langen Paketkopf und unterschiedliche, teilweise variabel lange Nutzlastsegmente besitzen (siehe Abbildung 6.2).

Der Paketkopf besteht aus einem acht Byte langen Feld `key`, das den Schlüssel zur Kon-

textzuordnung und das Return-Flag enthält. Ein Kommandofeld `cmd` enthält Flags zur Beschreibung von Verbindungsverwaltungs- bzw. Datentransfermodi sowie Information über die Protokollversion und das Format der PDU. Im folgenden nennen wir die Flags im Kommandofeld `cmd`, die zur Beschreibung des Modus dienen. `NOCHECK` legt fest, ob durch die Prüfsumme nur der Paketkopf oder auch die Nutzlast gesichert werden. `NOERR` zeigt an, ob Fehlerbehebung durch Datenwiederholung möglich ist. `MULTI` ist nur im Multicast-Modus gesetzt. `RES` bestimmt, ob der Empfänger in der Flußkontrolle nur Kredit für Daten geben darf, für die Empfangsspeicher vorhanden ist (Reservation Mode). Mit `SORT` kann man angeben, ob XTP-PDUs eine Priorität zugeordnet wird, die ihre Sendereihenfolge bestimmt. Durch `NOFLOW` wird festgelegt, ob der Kreditvergabemechanismus eingesetzt werden soll. `FASTNAK` zeigt an, ob der Empfänger von Daten Fehlermeldungen nur auf Anforderung des Senders oder auch von sich aus absetzen darf. Mit `SREQ` bzw. `DREQ` werden Bestätigungen bzw. Fehlermeldungen vom Empfänger angefordert, wobei er, wenn `SREQ` gesetzt ist, sofort antworten muß, während er, wenn `DREQ` gesetzt ist, erst nach Auslieferung der empfangenen Dateneinheiten an den Dienstnehmer reagiert. `WCLOSE` zeigt an, ob der Sender der PDU seinen Sendekanal geschlossen hat. `RCLOSE` in einer Verbindungsaufbau-PDU legt fest, ob ein Simplex- oder Duplex-Datenkanal aufgebaut wird. In allen anderen PDU-Formaten gibt es an, ob der Partner eine Anzeige zur Schließung des Sendekanals (`WCLOSE`) erhalten hat. Durch `END` kann das Verbindungsende angezeigt werden.

Betrachten wir nun die weiteren Felder des XTP-Paketkopfs. Das Feld `dlen` gibt die Länge des Nutzdatensegments an. Die Prüfsumme zur Erkennung von Bitfehlern ist in Feld `check` enthalten. In `sort` kann eine PDU-Priorität angegeben werden, die die Sendereihenfolge bestimmt, falls das Steuerflag `SORT` gesetzt ist. Das Feld `sync` enthält eine weitere Sequenznummer, die die PDUs zur Aushandlung neuer Datenraten eindeutig identifiziert[5]. Die Sequenznummer des Nutzdatenbyte, das als nächstes gesendet werden wird, ist in Feld `seq` enthalten.

Der Aufbau des Nutzdatensegments ist in den einzelnen PDU-Formaten unterschiedlich. XTP verwendet zwei Gruppen an PDU-Formaten. Die eine Gruppe umfaßt die drei PDU-Formate `CNTL`, `ECNTL` und `TCNTL` zur Übertragung von Protokollsteuerinformation wie Bestätigungen, Sendekrediten oder Fehlermeldungen. `CNTL` wird verwendet, wenn bei fehlerfreiem Ablauf Nutzdaten bestätigt und neue Kredite vergeben werden sollen. In Feld `rseq` ist die Sequenznummer des ersten Nutzdatenbyte, das noch nicht korrekt empfangen wurde, enthalten. In `alloc` ist das Kreditlimit angegeben. Nur Nutzdaten mit Sequenznummern, die kleiner als dieses Limit sind, dürfen gesendet werden. Das Feld `echo` enthält, wenn mit der PDU auf eine Anfrage des Verbindungspartners reagiert wird, denselben Wert wie das Feld `sync` in der PDU, mit der die Anfrage übertragen wurde. Das PDU-Format `ECNTL` wird verwendet, wenn Übertragungsfehler festgestellt wurden. Die Sendeinstanz kann eine Dateneinheit nur ein zweites Mal senden, wenn die Störung ihres Empfangs durch eine PDU dieses Formats angezeigt wird. Man kann außer `rseq` zusammenhängende Bereiche von Nutzdaten (Spans) angeben, die korrekt empfangen wurden. Nach dem Empfang dieser PDU werden alle Nutzdaten mit Sequenznummern ab `rseq` erneut gesendet, die nicht in einem Span enthalten sind. `ECNTL` enthält außer den Feldern `rseq`, `alloc` und `echo` ein Feld `nspan`, das die in der PDU enthaltene Anzahl an Spans angibt. Die Spans selbst sind in Feld `spans` gespeichert. Ein Span wird durch die Sequenznummer seines ersten Nutzdatenbyte und dem Nachfolger seines letz-

[5] In anderen PDU-Typen enthält `sync` den bei der letzten Verhandlung neuer Datenraten verwendeten Wert.

ten Nutzdatenbyte gekennzeichnet. Der Empfänger verwendet `ECNTL` immer, wenn er Lücken im Empfangsdatenstrom festgestellt hat. Das PDU-Format `TCNTL` wird zum Aushandeln neuer Datenraten verwendet. Dabei entscheidet das Steuerbit `SREQ`, ob es sich um eine Anfrage oder Bestätigung handelt. `TCNTL` enthält neben `rseq`, `alloc` und `echo` ein Feld `rsvd` nicht verwendeter, aber reservierter Information sowie das Feld `xkey`. Damit kann der Partner eines Verbindungsaufbauinitiators einen in seiner Station eindeutigen Schlüssel versenden, so daß die Kontextzuordnung von PDUs vereinfacht wird. Die Datenflußanzeige `Traffic Specifier` enthält Information über Datenraten, auf die hier nicht näher eingegangen wird.

Die zweite Gruppe an PDU-Formaten dient der Datenübertragung. Diese Formate sind `FIRST`, `DATA`, `JOIN` und `DIAG`. Das Format `FIRST` wird beim Verbindungsaufbau verwendet. Seine Nutzlast besteht aus einem Adreß-Segment, das die Adressen der beteiligten Stationen enthält, einem Feld, das zur Aushandlung einer initialen Datenflußrate dient, und gegebenenfalls Nutzdaten. Das Adreß-Segment besteht aus einer Segmentlängenanzeige `alen`, einem Feld `aformat`, mit dem Adreßformate (z.B. Internet Protocol-Adressen oder lokale Adressen) angezeigt werden, einer Adreßbereichsanzeige `adomain`, die von einigen Adreßformaten verwendet wird, und dem Feld `address`, das die Adressen von Initiator und dessen Partner enthält. Das PDU-Format `DATA` enthält nur Nutzdaten als Nutzlast. Das PDU-Format `JOIN` wird bei Multicast-Verbindungen zum Beitritt einer neuen Station verwendet und besteht aus einem Adreß-Segment und einer Datenflußanzeige. Das PDU-Format `DIAG` wird beim Auftreten ungewöhnlicher Ereignisse eingesetzt. Es besteht aus den Feldern `code` und `val`, durch die die Fehlerklasse angegeben wird, und einem Feld `message`, durch das der Fehler dem Dienstnehmer des Partners als Zeichenkette mitgeteilt werden kann.

6.2 Spezifikation des XTP-Protokolls

Bevor wir die Spezifikationen der XTP-Protokollmechanismen durch Instantiierung von Finite Abstract Protocol Mechanisms (FAPMs) des Frameworks (siehe Abschnitt 5.5) erstellen, definieren wir zunächst einen Datentyp `XTPpdu`, der das Format der XTP-PDUs beschreibt. Das PDU-Format von XTP wird durch die folgende tTLA$^+$-Spezifikation dargestellt:

XTPPDU

```
CONSTANT MODULE XTPPDU

  CONSTANTS
    Byte ≜ { x | x ∈ Nat ∧ x < 2^8 };    Byte
    DByte ≜ { x | x ∈ Nat ∧ x < 2^16 }; Doppel-Byte
    Word ≜ { x | x ∈ Nat ∧ x < 2^32 };   Wort = 4 Bytes
    DWord ≜ { x | x ∈ Nat ∧ x < 2^64 }; Doppelwort

    XTPpdu ≜ [ h : Header, p : Payload ];      XTP-PDU
    Header ≜ [ key : DWord, cmd : Word, dlen : Word, check : DByte, PDU-Kopf
               sort : DByte, sync : Word; seq : DWord ];
    Payload ≜ [ cs : ControlSegment, is : InformationSegment ];    PDU-Nutzinformation
    ControlSegment ≜ [ cc : CommonControl, ec : ErrorControl,      Steuersegment
                       tc : TrafficControl ];
    CommonControl ≜ [ rseq : DWord, alloc : DWord, echo : Word ]; CNTL
```

```
ErrorControl ≜ [ rseq : DWord, alloc : DWord, echo : Word,     ECNTL
                 nspan : Word, spans : { x | x ∈ Nat ∧ x < 2^(2^39) } ];
TrafficControl ≜ [ rseq : DWord, alloc : DWord, echo : Word, rvsd : Word,
                   xkey : DWord, ts : TrafficSpecifier ];      TCNTL

InformationSegment ≜ [ as : AddressSegment, ts : TrafficSpecifier,
  Informationssegment
                       ds : DataSegment, dias : DiagnosticSegment ];
AddressSegment ≜ [ alen : DByte, adomain : Byte, aformat : Byte,    Adresse
                   address : { x | x ∈ Nat ∧ x < 2^(2^19 − 2^5)} ];
DataSegment ≜ [ data : { x | x ∈ Nat ∧ x < 2^(2^35) } ];                Nutzdaten
DiagnosticSegment ≜ [ code : Word, val : Word,                          Diagnosedaten
                      message : { x | x ∈ Nat ∧ x < 2^(2^35 − 2^6)} ];

TrafficSpecifier ≜ [ tlen : DByte, service : Byte, tformat : Byte, Datenfluss
                     traffic : { x | x ∈ Nat ∧ x < 2^(2^19 − 2^5)} ];
```

```
cmdNOCHECK ( c : Word ) : {0,1} ≜ (c div 2^30) mod 2;      Zugriff auf Flags in cmd
cmdNOERR ( c : Word ) : {0,1} ≜ (c div 2^28) mod 2;
cmdMULTI ( c : Word ) : {0,1} ≜ (c div 2^27) mod 2;
cmdRES ( c : Word ) : {0,1} ≜ (c div 2^26) mod 2;
cmdSORT ( c : Word ) : {0,1} ≜ (c div 2^25) mod 2;
cmdNOFLOW ( c : Word ) : {0,1} ≜ (c div 2^24) mod 2;
cmdFASTNAK ( c : Word ) : {0,1} ≜ (c div 2^23) mod 2;
cmdSREQ ( c : Word ) : {0,1} ≜ (c div 2^22) mod 2;
cmdDREQ ( c : Word ) : {0,1} ≜ (c div 2^21) mod 2;
cmdRCLOSE ( c : Word ) : {0,1} ≜ (c div 2^20) mod 2;
cmdWCLOSE ( c : Word ) : {0,1} ≜ (c div 2^19) mod 2;
cmdEND ( c : Word ) : {0,1} ≜ (c div 2^17) mod 2;

cmdPFORMAT ( c : Word ) : { x | x ∈ Nat ∧ x < 2^5 } ≜ c mod 2^5; PDU-Format
DATA = 0; CNTL = 1; FIRST = 2; ECNTL = 3; TCNTL = 5; DIAG = 8;   PDU-Typ

getSPAN ( s : { x | x ∈ Nat ∧ x < 2^(2^39) }; n : Word;     Zugriff auf Span
          t : {"min","max"}) : DWord ≜
  IF (t = "min") THEN (s div 2^(n * 128)) mod 2^64
                 ELSE (s div 2^(n * 128 + 64)) mod 2^64;
getDATAPOS ( d : { x | x ∈ Nat ∧ x < 2^(2^35); n : DWord ) : Byte ≜
  (d div 2^(8 * n)) mod 2^8;                Zugriff auf Nutzdaten

keyRTN ( c : DWord ) : {0, 1} ≜ c div 2^63;
  Verbindungskennung bei Sender oder Empfaenger eindeutig?
keyKEY ( c : DWord ) : DWord ≜ c mod 2^63;   Verbindungskennung
```

Die Wertbelegungen, die Felder von einem Byte, zwei Bytes, vier Bytes oder acht Bytes Länge annehmen können, werden durch die Bezeichner `Byte`, `DByte`, `Word` bzw. `DWord` beschrieben.

`XTPpdu` bezeichnet das Format der PDUs von XTP. Es steht an der Wurzel einer Struktur aus tTLA$^+$-Feldern, die die unterschiedlichen Segmente der PDU-Strukturen enthält. Um

Parameter	Mechanismus	Belegung `0`	Belegung `1`
`NOERR`	Fehlerbehebung	ein	aus
`NOCHECK`	Bitfehlerprüfung	ganze PDU	nur PDU-Kopf
`NOFLOW`	Kreditvergabe	ein	aus
`RES`	Kreditvergabe	Kredit berücksichtigt die Größe des Empfangsspeichers	Kredit berücksichtigt nicht die Größe des Empfangsspeichers
`FASTNAK`	Fehlermeldung	initiiert durch den Sender	initiiert durch den Empfänger, wenn ein Übertragungsfehler erkannt wurde, oder durch den Sender

Tabelle 6.1: Bedeutung der Datentransfermodus-Parameter in FAPMs

einfacher auf Teile einzelner PDU-Felder zugreifen zu können, werden noch einige Zugriffsfunktionen angegeben. Zum Beispiel verweist `cmdNOCHECK(p.h.cmd)` auf das Flag `NOCHECK` im Protokollsteuerfeld `cmd` des Kopfes `h` von PDU `p`.

In Abschnitt 6.2.1 beschreiben wir zunächst den Aufbau der Datentransferkanäle und der XTP-Verbindungen. Anschließend gehen wir in Abschnitt 6.2.2 auf die Parameterbelegung der FAPMs des Datentransfers ein, deren Kombination in Abschnitt 6.2.3 erläutert wird. Die Parametrisierung sowie Kombination der Signalisierungsdatenkanal-FAPMs werden in den Abschnitten 6.2.4 und 6.2.5 angegeben. Die FAPMs der Intraconnection-Coordination beschreiben wir in den Abschnitten 6.2.6 und 6.2.7 und die der Interconnection-Coordination in den Abschnitten 6.2.8 und 6.2.9. Der Aufbau der AMCs wird in Abschnitt 6.2.10 und der der APMs in Abschnitt 6.2.11 vorgestellt.

6.2.1 Aufbau der Protokollspezifikation

Der Datentransfer kann unter Verwendung unterschiedlicher Datentransfermodi durchgeführt werden. Die Zusammensetzung der Protokollmechanismen hängt vom verwendeten Datentransfermodus ab, so daß für jeden Datentransfermodus eine eigene Datentransferkanalspezifikation erstellt werden muß.

Um den Spezifikationsaufwand gering zu halten, erstellen wir nicht für jeden Datentransferkanal eine eigene Spezifikation jedes Protokollmechanismus, sondern nur eine einzige für alle Kanäle. Die FAPMs erhalten deshalb neue Prozeßtypparameter, mit deren Hilfe sie an die Bedingungen der einzelnen Datentransfermodi angepaßt werden können. Diese Prozeßtypparameter werden `NOCHECK`, `NOERR`, `MULTI`, `RES`, `NOFLOW` und `FASTNAK` genannt und entsprechen den gleichnamigen Steuerflags in den PDUs von XTP (siehe Abschnitt 6.1.3 und Tabelle 6.1). Die Datenkanalspezifikationen werden durch die cTLA$^+$-Kombination parametrisierter FAPMs gebildet. Da wir nur Unicast-Verbindungen (`MULTI` nicht gesetzt) modellieren, gibt es theoretisch 32 unterschiedliche Datentransfermodi. Da jedoch einige Kombinationen der Flags nicht sinnvoll sind (z.B. `RES` und `NOFLOW` gesetzt; `FASTNAK` und `NOCHECK` gesetzt), müssen wir nur 18 Modi betrachten und als Datenkanäle spezifizieren. Außerdem bestehen alle Modi, die sich nur in der Belegung der Flags `NOCHECK` und `RES` unterscheiden, aus derselben

Zusammensetzung von FAPMs. Somit müssen effektiv nur sechs unterschiedliche Datentransferkanäle spezifiziert werden.

Zur Beschreibung der Signalisierungsdatenkanäle sind vier Kanalspezifikationen notwendig, da der Transfer der verschiedenen Signalisierungsdaten unter Verwendung unterschiedlicher Protokollfunktionen erbracht wird. Ein Kanal modelliert den Transport von PDUs für den Verbindungsaufbau. Signale zur Änderung des aktuellen Datentransfermodus und für den Abbau der Simplex-Datenkanäle werden über einen zweiten Kanal gesendet. Ein dritter Kanal dient zum Transfer von Signalen zum endgültigen Abbau der Verbindung. Über Kanäle des vierten Typs werden Signalisierungsdaten bei Verbindungsabbrüchen übertragen, die vom Diensterbringer initiiert werden.

Die Beschreibung einer XTP-Verbindung besteht aus der Kombination der Spezifikationen aller 18 Datentransfermodi und vier Signalisierungskanalspezifikationen. Insgesamt besteht eine Verbindungsbeschreibung aus 44 virtuellen Transportkanälen, da für jede der beiden Senderichtungen ein Kanal vorgesehen werden muß. Außerdem gehören zu der Verbindungsspezifikation die FAPMs der Intraconnection-Coordination, mit denen die Koordinierung der Signalisierungskanäle und der Datentransferkanäle modelliert wird.

Die Spezifikation des gesamten Transportsystems wird durch ein Prozeßfeld aus den Verbindungsspezifikationen gebildet. Zu diesem Feld werden die Interconnection-Coordination-FAPMs hinzukomponiert, durch die sichergestellt ist, daß alle PDUs in allen virtuellen Kanälen einer Verbindung eine einheitliche Kennung haben, so daß die virtuellen Kanäle auf einen realen physikalischen Kanal abgebildet werden können.

6.2.2 Instantiierung der FAPMs des Datentransfers

Vor Bildung der XTP-Spezifikationen müssen wir die Dienstprimitive an der Transportdienstschnittstelle definieren, da die XTP-Beschreibung [XTP95] selbst keine Schnittstellenbeschreibung enthält. Für den Datentransfer modellieren wir einen unbestätigten Datentransferdienst. Die Dienstelemente `Rq` und `In` erlauben die Sendung bzw. Auslieferung einzelner Nutzdatenbytes[6].

Wie schon in Abschnitt 6.2.1 erwähnt wurde, bilden wir für jeden Protokollmechanismus nur eine Spezifikation, die in unterschiedlichen Spezifikationen der Datentransfermodi eingesetzt wird. Um die in den einzelnen Modi voneinander abweichenden Ausprägungen der Protokollmechanismen beschreiben zu können, führen wir in den Instanzen neue Prozeßtypparameter `NOERR`, `NOCHECK`, `NOFLOW`, `RES` und `FASTNAK` ein, die jeweils die Werte `0` oder `1` annehmen können. Die Namen der Parameter beziehen sich auf die gleichnamigen Steuerflags in den XTP-PDUs zur Modusanzeige (siehe Tabelle 6.1).

Vor der Betrachtung der Parameterersetzungen in den einzelnen FAPMs des Datentransfers (siehe Abschnitt 5.5.1 sowie 5.3.1 und 5.3.2) gehen wir zunächst auf die Prozeßtypparameter ein, die in (fast) allen FAPMs verwendet werden:

```
pdu ≙ XTPpdu;
pci ≙ XTPpdu;
spci : SET[ XTPpdu → XTPpdu] ≙ [ x ∈ XTPpdu ↦ x ]
usd ≙ Byte;
fkey ≙ DWord ∪ {"notsent"};
```

[6] Weitere Dienstelemente zur Verbindungsverwaltung werden in Abschnitt 6.2.4 eingeführt.

```
fpkey ≜ DWord ∪ {"notsent"};
mseg ≜ 1;
skk ≜ SET[ x ∈ DWord ∪ {"notsent"} ↦ x ];
skn ≜ SET[ DWord ∪ {"notsent"} ↦ 0 ];
skm ≜ SET[ DWord ∪ {"notsent"} ↦ 1 ];
usdsize ≜ SET[ usd ↦ 1 ];
usdsplit : SET[ (Nat × usd) → usd] ≜ [ (n,d) ∈ (Nat × usd) ↦ d ];
encpdu ≜ { p ∈ XTPpdu | ( cmdNOCHECK(p.h.cmd) = 0 ∧
                          p.h.check = xsum(p.h.dlen + 32, p) ) ∨
                        ( cmdNOCHECK(p.h.cmd) = 1 ∧
                          p.h.check = xsum(32, p.h) ) };
ir ≜ 2^64;
```

Der formale Parameter `pdu`, der das PDU-Format beschreibt, wird durch den oben angegebenen Datentyp `XTPpdu` ersetzt. `pci` beschreibt die Protokollsteuerinformation in PDUs. Da in XTP-PDUs alle Felder mit Ausnahme der Datensegmente Protokollsteuerinformation (PCI) enthalten, ersetzen wir zur Vereinfachung `pci` ebenfalls durch `XTPpdu`. Die Abbildung `spci`, mit der eine Referenz auf die PCI einer PDU gegeben wird, ist deshalb die identische Abbildung. `usd` beschreibt die Menge aller möglichen Nutzdatenpakete, die als Einheit übertragen werden kann. Da XTP Datentransfer auf Byte-Strömen vorsieht, ist jedes Nutzdatenbyte eine eigenständige Dateneinheit, die eine eigene Sequenznummer erhält. `usd` wird somit durch `Byte` ersetzt. XTP verwendet acht Byte lange Sequenznummern, so daß wir `fkey` durch die Menge `DWord` sowie den Bezeichner `"notsent"` für Phantompakete ersetzen. Bytes als Übertragungseinheiten werden nicht segmentiert, so daß der Parameter `fpkey` ebenfalls durch `DWord ∪ "notsent"` ersetzt wird. Die Parameter `mseg`, `skk`, `skn`, `skm`, `usdsize` und `usdsplit` (siehe auch Abschnitt 5.3.1) werden durch Funktionen ersetzt, die Daten-Segmentierung ausschließen. Die Menge `encpdu` modelliert die PDUs, die vom Empfänger als fehlerfrei erkannt werden. Das sind in XTP die PDUs `p`, deren Prüfsumme `p.h.check` dem Ergebnis der Prüffunktion `xsum` entspricht, die in [XTP95] aufgelistet ist. Je nach Belegung des Steuerflags `NOCHECK` (`cmdNOCHECK(p.h.cmd)`) wird entweder die ganze PDU `p` oder nur der Kopf `p.h` überprüft. Der Parameter `ir` gibt die Anzahl an Sequenznummern der übertragenen Daten an. Da die Sequenznummern in XTP acht Bytes lang sind, instantiieren wir ihn mit dem Wert 2^{64}.

Wir beginnen mit der Beschreibung der FAPMs *SBufferKey* und *RBufferKey*, die die Verwaltung der Sequenznummern der Dateneinheiten modellieren. Außer den oben genannten Prozeßtypparametern wird nur `skey` ersetzt. `skey` beschreibt die Position der Sequenznummern der übertragenen Dateneinheiten innerhalb der PCI einer PDU. `skey` besitzt den Wert `"<<>>"`, wenn keine Daten in der PDU übertragen werden:

```
skey ≜
  [ p ∈ XTPpdu ↦
    IF ( cmdNOERR(p.h.cmd) = NOERR ∧
         cmdNOCHECK(p.h.cmd) = NOCHECK ∧
         cmdNOFLOW(p.h.cmd) = NOFLOW ∧
         cmdRES(p.h.cmd) = RES ∧
         cmdFASTNAK(p.h.cmd) = FASTNAK )
   THEN
      CASE cmdPFORMAT(p.h.cmd) = DATA →
```

```
            { b ∈ fpkey | between(p.h.seq, b, (p.h.seq + p.h.dlen) mod 2^64)},
        cmdPFORMAT(p.h.cmd) = FIRST ∧
          p.h.dlen > p.p.is.alen - p.p.is.tlen →
            { b ∈ fpkey | b < p.h.dlen - p.p.is.alen - p.p.is.tlen },
        OTHERWISE → "≪ ≫"
    ELSE "≪ ≫" ];
```

Nutzdaten können nur in Daten-PDUs (Format `DATA`) oder in Verbindungsaufbau-PDUs (Format `FIRST`) übertragen werden. In einer Daten-PDU besteht das ganze Nutzlastsegment aus einer zusammenhängenden Folge von Nutzdatenbytes. Das erste Byte der Folge wird durch das Feld `p.h.seq` im Paketkopf angezeigt. Die Länge der Datenfolge entspricht der Länge des Nutzlastsegments, die in Feld `p.h.dlen` angegeben ist. In einer Daten-PDU befinden sich also genau die Datenbytes mit Sequenznummern zwischen `p.h.seq` und `p.h.seq + p.h.dlen - 1`. Die erste in der Verbindungsaufbau-PDU enthaltene Dateneinheit hat immer die Sequenznummer `0`. Die Anzahl der übertragenen Daten wird aus `p.h.dlen`, der Länge `p.p.is.alen` des Adreß-Segments und der Länge `p.p.is.tlen` der Datenflußanzeige bestimmt. In allen anderen PDUs und auch bei Verbindungsaufbau-PDUs ohne Datensegment wird `skey` der Wert `"<<>>"` zugewiesen, da dort keine Nutzdaten enthalten sind. Auch PDUs, deren Steuerflags nicht mit den Belegungen der Parameter `NOERR`, `NOCHECK`, `NOFLOW`, `RES` und `FASTNAK` übereinstimmen, erhalten den Wert `"<<>>"`, da sie nicht den Transfer von Daten auf dem durch die Instanz des FAPMs modellierten Datenkanal beschreiben. Die Parameter `mb`, `rcc` und `rcu` bestimmen die Größe von Sendepuffer und Empfangspuffer. Sie sind implementierungsabhängig und werden von uns nicht ersetzt.

Als Beispiel zeigen wir das FAPM *XTPSBufferKey*, das mit Hilfe des cTLA$^+$-Überführungswerkzeugs [Hey95] aus dem in Abschnitt 5.5.1 aufgeführten FAPM *SBufferKey* erzeugt wurde[7]:

```
PROCESS XTPSBufferKey ( mb : Nat mb ist der einzige nicht ersetzte Parameter;
                        NOERR, NOCHECK, NOFLOW, RES, FASTNAK : {0, 1}
                        durch diese Parameter wird der beschriebene Datentransfermodus angezeigt )
IMPORT Constants;
BODY
  VARIABLES
    cRq : DWord ∪ {"notsent"} fkey;
    sbk : SUBSET(DWord ∪ {"notsent"}) SUBSET(fpkey);
  INIT ≜ cRq = 0 ∧ sbk = ∅;
  ACTIONS
    Rq (krq : DWord ∪ {"notsent"}) ≜
      krq = cRq ∧
      sbk' = (sbk \ { (cRq + 1) mod 2^64 ir }) ∪ { cRq } ∧
XTP kennt keine Segmentierung. Deshalb wird in Aktion Rq der Menge sbk die Sequenznummer krq
der Dateneinheit hinzugefuegt.
      cRq' = inc(cRq);
    MRq ( krq : SUBSET(DWord) ∪ {"notsent","≪ ≫"}; p : XTPpdu ) ≜
      ( ( cmdNOCHECK(p.h.cmd) = 0 ∧ p.h.check = xsum(p.h.dlen + 32, p) ) ∨
        ( cmdNOCHECK(p.h.cmd) = 1 ∧ p.h.check = xsum(32, p.h) ) )
p ∈ encpdu (als Schreibvereinfachung wurde p in { k : P(k) } durch P(p) ersetzt) ∧
      krq = IF ( cmdNOERR(p.h.cmd) = NOERR ∧ cmdNOCHECK(p.h.cmd) = NOCHECK ∧
```

[7] Als cTLA$^+$-Prozeßoperation wurde die Prozeßkomposition verwendet (siehe Abschnitt 4.3.2). In *XTPSBufferKey* sind die Konstrukte des Original-FAPM als Kommentare in kursiver Schrift angegeben.

```
                cmdNOFLOW(p.h.cmd) = NOFLOW  ∧  cmdRES(p.h.cmd) = RES ∧
                cmdFASTNAK(p.h.cmd) = FASTNAK )
            THEN
              CASE cmdPFORMAT(p.h.cmd) = DATA →
                     { b ∈ DWord ∪ {"notsent"} |
                         between(p.h.seq, b, (p.h.seq + p.h.dlen) mod 2^64)},
                   cmdPFORMAT(p.h.cmd) = FIRST  ∧
                     p.h.dlen > p.p.is.alen - p.p.is.tlen →
                       { b ∈ DWord ∪ {"notsent"} |
                          b < p.h.dlen - p.p.is.alen - p.p.is.tlen },
                   OTHERWISE → "≪ ≫"
            ELSE "≪ ≫"
krq = skey[spci[p] ] ∧
      ( krq = "≪ ≫" ∨ ( krq ⊆ sbk  ∧  card(krq) ≤ mb ) ) ∧
      cRq' = cRq  ∧  sbk' = sbk;
  END
```

Die Aktion `Rq` modelliert die Sendung einzelner Nutzdatenbytes durch den Sender und beschreibt damit den Sendevorgang in einem byteorientierten Datenstrom. Durch den Zähler `cRq` wird jedem Byte inkrementell eine Sequenznummer zugeordnet. Die neu vergebene Sequenznummer wird in den Puffer `sbk` eingefügt und die veraltete Sequenznummer `(cRq + 1) mod DWord` daraus gelöscht. An der Schnittstelle zum Basiskanal M (Aktion `MRq`) werden Eigenschaften der gesendeten PDUs festgelegt. Wenn der in einer PDU angezeigte Datentransfermodus mit dem des Basiskanals übereinstimmt, müssen die Sequenznummern der durch die PDU gesendeten Datenbytes im Sendepuffer `sbk` enthalten sein. Außerdem darf die Gesamtzahl der Sequenznummern in der PDU nicht den Wert `mb` überschreiten.

Als nächstes parametrisieren wir die FAPMs *SBufferUsd* und *RBufferUsd*, die die Nutzdatenspeicher der Sender- und Empfängerinstanzen spezifizieren. Hier wird über die oben beschriebenen Parametrisierungen hinaus der Parameter `susd` ersetzt, durch den die Position einer Nutzdateneinheit mit einer bestimmten Sequenznummer `k` in einer PDU `p` referenziert wird:

```
susd ≙ [ (p,k) ∈ (XTPpdu × (DWord ∪ {"notsent"})) ↦
           CASE cmdPFORMAT(p.h.cmd) = DATA →
                  getDATAPOS(p.p.is.ds.data, (k - p.h.seq) mod 2^64),
                cmdPFORMAT(p.h.cmd) = FIRST →
                  getDATAPOS(p.p.is.ds.data, k) ];
```

Die Funktion `getDATAPOS(`n`)` dient der Referenzierung des n-ten Datenbyte im Datensegment der XTP-PDUs. In PDUs des Formats `DATA` besitzt die erste Dateneinheit im Datensegment die Sequenznummer `p.h.seq`. Die Bytes sind im Datensegment in aufsteigender Reihenfolge gespeichert. Somit befindet sich das Byte mit der Sequenznummer `k` im Datensegment an der Position `(k - p.h.seq) mod` 2^{64}. Bei Verbindungsaufbau-PDUs wird ab Sequenznummer `0` übertragen, und die Position jedes Nutzdatenbyte entspricht seiner Sequenznummer.

`sackm` und `sacko` sind die zu instantiierenden Parameter der FAPMs *SAcknowledge* und *RAcknowledge*, die den Bestätigungsmechanismus empfangener Daten modellieren, sowie des FAPMs *SCapacity*, durch das das Sendepufferlimit beschrieben wird. Durch `sackm` wird die Sequenznummer des Vorgängers der ersten Dateneinheit beschrieben, die noch nicht korrekt beim Empfänger eingetroffen ist. Wenn der Empfänger eine Lücke verlorengegangener oder

verfälschter Dateneinheiten erkannt hat, kann er die Sequenznummern der nach dieser Lücke korrekt empfangenen Dateneinheiten im Feld `sacko` bestätigen:

```
sackm ≙ [ p ∈ XTPpdu ↦
          IF ( cmdNOERR(p.h.cmd) = NOERR ∧
               cmdNOCHECK(p.h.cmd) = NOCHECK ∧
               cmdNOFLOW(p.h.cmd) = NOFLOW ∧
               cmdRES(p.h.cmd) = RES ∧
               cmdFASTNAK(p.h.cmd) = FASTNAK )
          THEN
            CASE cmdPFORMAT(p.h.cmd) = CNTL → (p.p.cc.rseq - 1) mod 2^64,
                 cmdPFORMAT(p.h.cmd) = ECNTL ∧ NOERR = 0 →
                   (p.p.ec.rseq - 1) mod 2^64,
                 cmdPFORMAT(p.h.cmd) = TCNTL ∧ SREQ(p.h.cmd) = 0 →
                   (p.p.tc.rseq - 1) mod 2^64,
                 OTHERWISE → "≪ ≫"
          ELSE "≪ ≫" ];
sacko ≙ [ p ∈ XTPpdu ↦
          IF ( cmdNOERR(p.h.cmd) = NOERR ∧
               cmdNOCHECK(p.h.cmd) = NOCHECK ∧
               cmdNOFLOW(p.h.cmd) = NOFLOW ∧
               cmdRES(p.h.cmd) = RES ∧
               cmdFASTNAK(p.h.cmd) = FASTNAK )
          THEN
            CASE cmdPFORMAT(p.h.cmd) = ECNTL ∧
                   NOERR = 0 →
                     { k ∈ pkey | ∃ n ∈ Word :
                         n < p.p.ec.nspan ∧
                         between(getSPAN(p.p.ec.spans, n, "min"), k,
                                 getSPAN(p.p.ec.spans, n, "max")) },
                 ( cmdPFORMAT(p.h.cmd) = CNTL ∨
                   ( cmdPFORMAT(p.h.cmd) = TCNTL ∧ cmdSREQ(p.h.cmd) = 0 ) )
                   → ∅,
                 OTHERWISE → "≪ ≫"
          ELSE "≪ ≫" ];
```

Die Sequenznummer des ersten nicht empfangenen Nutzdatenbyte ist in den PDU-Formaten `CNTL`, `ECNTL` und `TCNTL` in Feld `rseq` enthalten. Somit kann `sackm` durch den Vorgänger dieser Sequenznummer ersetzt werden. In `TCNTL` wird der Wert allerdings nur verwendet, wenn die PDU zur Bestätigung der Verhandlung einer Datenrate dient[8]. Daten, deren Sequenznummern größer als `rseq` sind, können nur durch PDUs des Formats `ECNTL` bestätigt werden. Dort werden alle Nutzdatenbytes bestätigt, deren Sequenznummer in einem Span enthalten sind. `sacko` enthält somit alle Sequenznummern, die in Spans liegen.

Die Übertragung von Fehlermeldungen wird durch die FAPMs *SReject* und *RReject* modelliert, in denen der Parameter `snak` instantiiert wird, der die Sequenznummern nicht korrekt empfangener Dateneinheiten in einer PDU beschreibt:

```
snak ≙ [ p ∈ XTPpdu ↦
           IF ( cmdPFORMAT(p.h.cmd) = ECNTL ∧
                NOERR = 0 ∧
```

[8] Das Steuerbit `SREQ` ist in diesem Fall nicht gesetzt.

```
                cmdNOERR(p.h.cmd) = 0 ∧
                cmdNOCHECK(p.h.cmd) = NOCHECK ∧
                cmdNOFLOW(p.h.cmd) = NOFLOW ∧
                cmdRES(p.h.cmd) = RES ∧
                cmdFASTNAK(p.h.cmd) = FASTNAK )
            THEN { k ∈ DWord ∪ {"notsent"} |
                    between(p.p.ec.rseq, k, getSPAN(p.p.ec.spans, 0, "min")) ∨
                    ∃ n ∈ Word :
                      0 < n < p.p.ec.nspan ∧
                      between(getSPAN(p.p.ec.spans, n - 1, "max"), k,
                              getSPAN(p.p.ec.spans, n, "min")) }
            ELSE "≪ ≫" ];
```

Nicht empfangene Dateneinheiten werden in XTP durch PDUs des Formats `ECNTL` gemeldet, die man nur verwendet, wenn Fehlerbehebungsmaßnahmen vorgesehen sind (`NOERR = 0`). In `ECNTL` werden alle Sequenznummern als fehlerhaft zurückgewiesen, die größer oder gleich dem Feld `p.p.ec.rseq` sind und nicht in einem Span liegen[9].

Die Vergabemechanismen für Sendekredite werden durch die FAPMs *SCredit* und *RCredit* modelliert. Der Parameter `scre` beschreibt die Position der Sequenznummer innerhalb einer PDU, die den Kreditrahmen angibt. Das entspricht genau dem Vorgänger der in Feld `alloc` in den XTP-PDU-Formaten `CNTL`, `ECNTL` und `TCNTL` angegebenen Sequenznummer:

```
scre ≙ [ p ∈ XTPpdu ↦
            IF ( cmdNOERR(p.h.cmd) = NOERR ∧
                 cmdNOCHECK(p.h.cmd) = NOCHECK ∧
                 cmdNOFLOW(p.h.cmd) = NOFLOW ∧
                 cmdRES(p.h.cmd) = RES ∧
                 cmdFASTNAK(p.h.cmd) = FASTNAK )
         THEN CASE cmdPFORMAT(p.h.cmd) = CNTL →
                     IF (NOFLOW = 0)
                     THEN (p.p.cs.cc.alloc - 1) mod DWord
                     ELSE p.p.cc.rseq,
                  cmdPFORMAT(p.h.cmd) = ECNTL →
                     IF (NOFLOW = 0)
                     THEN (p.p.cs.ec.alloc - 1) mod DWord
                     ELSE p.p.ec.rseq,
                  cmdPFORMAT(p.h.cmd) = TCNTL ∧
                    cmdSREQ(p.h.cmd) = 0 →
                     IF (NOFLOW = 0)
                     THEN (p.p.cs.tc.alloc - 1) mod DWord
                     ELSE p.p.tc.rseq,
                  OTHERWISE → "≪ ≫"
         ELSE "≪ ≫" ];
mo ≙ IF (RES = 1) THEN 0 ELSE momax;
mr ≙ 0;
```

Die Mechanismen der Kreditvergabe werden nur verwendet, wenn die Flußkontrolle nicht abgeschaltet wurde (`NOFLOW = 0`). In den Spezifikationen der Transfermodi mit `NOFLOW = 1`

[9] Wir erläutern in Abschnitt 6.5.3, daß die Beschreibung des Fehlermeldungsmechanismus in der XTP-Beschreibung [XTP95] unzureichend ist. In dieser Spezifikation modellieren wir die erste der beiden Lösungsansätze, die in Abschnitt 6.5.3 dargestellt werden.

sind *SCredit* und *RCredit* deshalb nicht enthalten. Da der Parameter `scre` jedoch auch in den Liveness-FAPMs verwendet wird (siehe Abschnitt 5.3.2), geben wir oben auch für den Fall der Abschaltung der Kreditvergabe eine Ersetzung an, so daß wir *SCredit* und *RCredit* auch zur Modellierung der Livenesseigenschaften einsetzen können. Für die Verwendung in den Liveness-FAPMs reicht die Sequenznummer des Feldes `rseq` als Kreditlimit aus. Ein weiterer Parameter im FAPM *RCredit* ist `mo`, der festlegt, ob und bis zu welcher Höhe Kredit für Nutzdaten gegeben werden darf, für die kein Empfangspuffer vorhanden ist. Im Reservation Mode (`RES = 1`) darf Kredit nur für Daten gegeben werden, die vom Empfänger gespeichert werden können. Sonst wird er durch eine Schranke `momax`, die als neuer Prozeßtypparameter angegeben wird, begrenzt. `mr` gibt eine Grenze für nachträgliche Kreditverminderungen (Reneging) an. Da in [XTP95] nicht erwähnt wird, daß Kredite nachträglich reduziert werden dürfen, lassen wir Reneging nicht zu (`mr` $\triangleq$0).

Durch die FAPMs *STriggerAck*, *RTriggerAck*, *STriggerCredit* und *RTriggerCredit* wird modelliert, daß eine Station Bestätigungen und Fehlermeldungen bzw. Kreditvergaben nur nach Anforderung ihres Partners versenden darf. In XTP werden diese Anforderungen durch die Steuerflags `SREQ` und `DREQ` dargestellt. Wir können also die Parameter `stack` und `stcre`, die die Anforderungsinformation in PDUs beschreiben, folgendermaßen ersetzen:

```
stack ≙ [ p ∈ XTPpdu ↦
            IF ( ( cmdSREQ(p.h.cmd) = 1 ∨ cmdDREQ(p.h.cmd) = 1 ∨
                   FASTNAK = 1 ) ∧
                 cmdNOERR(p.h.cmd) = NOERR ∧
                 cmdNOCHECK(p.h.cmd) = NOCHECK ∧
                 cmdNOFLOW(p.h.cmd) = NOFLOW ∧
                 cmdRES(p.h.cmd) = RES ∧
                 cmdFASTNAK(p.h.cmd) = FASTNAK )
            THEN "on" ELSE "off" ];
stcre ≙ [ p ∈ XTPpdu ↦
            IF ( ( cmdSREQ(p.h.cmd) = 1 ∨ cmdDREQ(p.h.cmd) = 1 ∨
                   FASTNAK = 1 ) ∧
                 cmdNOERR(p.h.cmd) = NOERR ∧
                 cmdNOCHECK(p.h.cmd) = NOCHECK ∧
                 cmdNOFLOW(p.h.cmd) = NOFLOW ∧
                 cmdRES(p.h.cmd) = RES ∧
                 cmdFASTNAK(p.h.cmd) = FASTNAK )
            THEN "on" ELSE "off" ];
```

Obwohl die FAPMs nur in Modi eingesetzt werden, in denen `FASTNAK = 0` gilt, müssen wir die Liveness-FAPMs berücksichtigen und geben deshalb auch eine Instantiierung für `FASTNAK = 1` an.

Die Ordnung des Ausgabedatenstroms wird durch die FAPMs *RGaps*, *RDuplicates*, *RReorderings* und *RPhantoms* spezifiziert. Lücken im Datenstrom sollen nur in Datentransfermodi zulässig sein, die keine Fehlerbehebung unterstützen (`NOERR = 0`). Die Auslieferung von duplizierten Daten, vertauschten Daten und Phantomen werden verboten:

```
td ≙ 0;
tg ≙ IF NOERR = 0 THEN 0 ELSE "infty";
tr ≙ 0;
pd ≙ SUBSET(Byte);
```

FAPM	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 1 0	NOERR NOFLOW FASTNAK
SBufferKey	⋆	⋆	⋆	⋆	⋆	⋆	
RBufferKey	⋆	⋆	⋆	⋆	⋆	⋆	
SBufferUsd	⋆	⋆	⋆	⋆	⋆	⋆	
RBufferUsd	⋆	⋆	⋆	⋆	⋆	⋆	
SCapacity	⋆	⋆	⋆	⋆	⋆	⋆	
SAcknowledge	⋆	⋆	⋆	⋆	⋆	⋆	
RAcknowledge	⋆	⋆	⋆	⋆	⋆	⋆	
SReject	⋆	⋆	⋆	⋆	⋆	⋆	
RReject	⋆	⋆	⋆	⋆			
SCredit	⋆	⋆			⋆		
RCredit	⋆	⋆			⋆		
STriggerAck	⋆		⋆		⋆	⋆	
RTriggerAck	⋆		⋆		⋆	⋆	
STriggerCredit	⋆				⋆		
RTriggerCredit	⋆				⋆		
RGaps	⋆	⋆	⋆	⋆			
RDuplicates	⋆	⋆	⋆	⋆	⋆	⋆	
RReorderings	⋆	⋆	⋆	⋆	⋆	⋆	
RPhantoms	⋆	⋆	⋆	⋆	⋆	⋆	
SSendInOrder	⋆	⋆	⋆	⋆	⋆	⋆	
RVoluntaryGaps	⋆	⋆	⋆	⋆	⋆	⋆	
SLiveMRq	⋆	⋆	⋆	⋆	⋆	⋆	
RLiveARq	⋆	⋆	⋆	⋆	⋆	⋆	
RLiveIn	⋆	⋆	⋆	⋆	⋆	⋆	

Tabelle 6.2: Komposition der Datentransfer-FAPMs

Wir haben nun alle Parameterersetzungen erläutert, die für die Entwicklung der XTP-FAPMs des Datentransfers notwendig sind. In den oben noch nicht genannten FAPMs *SSend-InOrder*, *RVoluntaryGaps*, *SLiveMRq*, *RLiveARq* und *RLiveIn* sind nur Prozeßtypparameter definiert, die entweder nicht ersetzt werden oder für die wir oben bereits eine Ersetzung angegeben haben.

6.2.3 Komposition der FAPMs des Datentransfers

Die Zuordnung der FAPMs zu den Datentransfermodusspezifikationen ist sehr einfach. Wie schon in Abschnitt 6.2.1 erörtert wurde, existieren 18 verschiedene Datentransfermodi. Da die Werte der Prozeßtypparameter `NOCHECK` und `RES` (siehe Tabelle 6.1) keinen Einfluß auf die Zusammensetzung der Kombinationen haben, durch die Modusspezifikationen erzeugt werden, reicht die Erstellung von sechs verschiedenen Kombinationen und damit sechs Datentransfermodusspezifikationstypen aus, die von den Parametern `NOERR`, `NOFLOW` und `FASTNAK`

Bezeichner	Beschreibung	Bestätig.	Dienstg.
`"FirstDuplex"`	Aufbau einer Duplex-Verbindung	nein	ja
`"FirstSimplex"`	Aufbau einer Simplex-Verbindung vom Dienstinitiator zu dessen Partner	nein	ja
`"NewQ"`	Umschaltung des Datentransfermodus	implizit	ja
`"WClose"`	Beendigung des Simplex-Kanals zwischen dem Dienstinitiator und seinem Partner	implizit	nein
`"Final"`	Ordnungsgemäßer Abbau der Verbindung, der nur initiiert werden darf, wenn der Simplex-Kanal vom Partner zum Initiator geschlossen ist.	nein	nein
`"UAbort"`	Abbruch der Verbindung durch einen Dienstnehmer	nein	nein
Provider-Abort	Abbruch der Verbindung durch den Diensterbringer	beidseitig	nein

Tabelle 6.3: Dienste der Intraconnection-Coordination

abhängen. In Tabelle 6.2 wird die Zuordnung der einzelnen FAPMs zu den sechs Kombinationen angegeben.

Wenn keine Fehlerbehebung verwendet wird (`NOERR = 1`), ist das FAPM *RReject*, das das Absetzen von Fehlermeldungen beschreibt, nicht notwendig. Außerdem verwenden wir das FAPM *RGaps* nicht, weil durch Verluste entstehende Lücken im Datenstrom nicht durch Übertragungswiederholungen behoben werden können und deshalb vom Protokoll toleriert werden müssen. Hingegen muß das FAPM *SReject* in den Beschreibungen vorhanden sein, da es das mehrfache Senden einer Nutzdateneinheit ausschließt. Schaltet man die Kreditvergabe aus (`NOFLOW = 1`), werden die FAPMs *SCredit*, *RCredit*, *STriggerCredit* und *RTriggerCredit* obsolet. Läßt man direkte Reaktionen auf Fehlermeldungen zu (`FASTNAK = 1`), wird auf die FAPMs *STriggerAck*, *RTriggerAck*, *STriggerCredit* und *RTriggerCredit* verzichtet.

6.2.4 Instantiierung der FAPMs der Signalisierungskanäle

Wie bei der Entwicklung der Spezifikationen der Datentransferkanäle müssen wir zunächst eine Transportdienstschnittstelle festlegen. Dabei orientieren wir uns an den Diensteigenschaften, die in Hochleistungstransportprotokollen üblich sind (siehe Abschnitt 2.2.3). Zum Beispiel verwenden wir nur implizite Dienstbestätigungen, da dadurch Laufzeitverzögerungen aufgrund verspäteter Reaktionen der Empfängerstation ausgeschlossen werden. Da XTP Mechanismen zur Aushandlung der Dienstgüte durch die Protokollinstanzen vorsieht, so daß der Dienstnehmer nur informiert, aber nicht konsultiert werden muß (vgl. [XTP95]), sind implizite Bestätigungen möglich. Die Dienstelemente sind in Tabelle 6.3 aufgeführt. Vorgesehen sind zwei nicht bestätigte Dienste `"FirstDuplex"` und `"FirstSimplex"` zum Aufbau von Duplex- bzw. Simplex-Verbindungen und ein implizit bestätigter Dienst `"NewQ"` zur Umschaltung des Datentransfermodus. Diese Dienste enthalten Dienstgüteparameter. In unserer funktionalen Beschreibung reicht es aus, den vom Dienstnehmer gewünschten Datentransfermodus

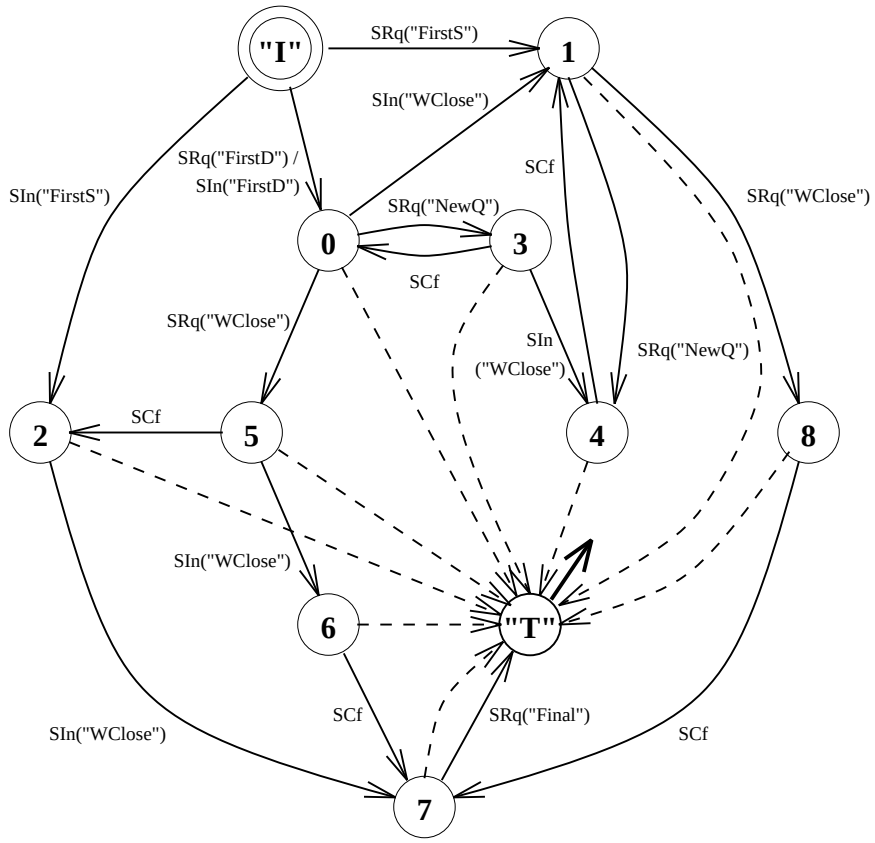

Abbildung 6.3: Ablaufverhalten der XTP-Dienstelemente

als Dienstgüte anzugeben. Weitere Dienste ohne Dienstgüteinformation sind der bestätigte Dienst `"WClose"`, mit dem man den Datentransfer für jede Übertragungsrichtung separat abbauen kann, und die unbestätigten Dienste `"Final"` zum ordnungsgemäßen Verbindungsabbau und `"UAbort"` zum Verbindungsabbruch. Weiterhin existiert noch ein Dienst zum Verbindungsabbruch durch den Diensterbringer (`Provider-Abort`, siehe auch Abschnitt 5.2.3).

Zur Spezifikation der Dienstschnittstelle gehört auch die Beschreibung des vom Dienstnehmer geforderten Ablaufverhaltens der Dienstelemente, die in Abbildung 6.3 als Graph angegeben ist. Zu Anfang wird eine Verbindung aufgebaut, was in XTP stets implizit erfolgt. Dann können Daten übertragen und die Datentransfermodi geändert werden. Nach Ende des Datentransfers schließen die Stationen die Datentransferkanäle voneinander separat und terminieren anschließend die Verbindung. Ein Verbindungsabbruch kann initiiert werden, solange die Verbindung aktiv ist. Dieses Ablaufverhalten wird durch das FAPM *UserBehaviour*

(siehe Abschnitt 6.2.6) spezifiziert.

Zur Spezifikation der Übertragung der PDUs, mit denen die Dienste der Verbindungsverwaltung erbracht werden, verwenden wir wieder die FAPMs des Datentransfers. Wie schon in Abschnitt 6.2.1 erwähnt wurde, benötigen wir vier verschiedene Kanaltypen, da sich die Funktionalität der Dienste vor allem in Hinsicht auf die Fehlerbehebungs- und Bestätigungsmechanismen stark voneinander unterscheidet. In den FAPMs führen wir dazu einen Parameter `ChType` ein, der die Werte `"First"`, `"Active"` oder `"Last"` annehmen kann. In `"First"` werden die Dienstelemente `"FirstDuplex"` und `"FirstSimplex"` übertragen. Beim Verbindungsaufbau ist eine durch explizite Zurückweisungen gesteuerte Fehlerbehebung nicht möglich, da die Partnerinstanz keine Kenntnis über verlorengegangene PDUs besitzen kann. Unsere Spezifikation sieht deshalb vor, daß Verbindungsaufbau-PDUs solange wiederholt werden dürfen, bis eine Reaktion durch den Partner erfolgt[10].

Die Dienste für die Veränderung des Datentransfermodus (`"NewQ"`) und des Simplex-Kanalabbaus (`"WClose"`) werden über den Kanal `"Active"` übertragen, dessen Protokollfunktionen im wesentlichen denjenigen von `"First"` entsprechen. Nur wird im Gegensatz zu `"First"` und den Datentransferkanälen jedem Signal in jeder Wiederholung eine eigene Sequenznummer zugeordnet. Die Anzahl der Wiederholungen eines Signals darf eine bestimmte Schranke nicht überschreiten. Die Übertragung gilt als abgeschlossen, wenn eine Bestätigung mit der zuletzt gesendeten Sequenznummer eintrifft. Diese Protokollfunktion kann durch das in Kapitel 5 vorgestellte Framework nicht spezifiziert werden, da dort einer Nutzdateneinheit immer nur eine Sequenznummer zugewiesen wird. Wir zeigen jedoch, daß man das Framework sehr einfach erweitern kann, um auch diesen Kanaltyp modellieren zu können.

Die Dienstelemente der Verbindungsterminierung `"Final"` und `"UAbort"` werden über den Kanal `"Last"` gesendet, der keine Bestätigungsmitteilungen kennt. PDUs werden wiederholt, wenn PDUs eintreffen, die vom Verbindungspartner erzeugt wurden und darauf hinweisen, daß die Terminierungsnachricht verlorengegangen sein könnte.

Ein vierter Kanal dient der Übertragung von Signalen eines vom Diensterbringer initiierten Verbindungsabbaus.

Bevor wir mit der Beschreibung der Parameterersetzungen beginnen, führen wir zunächst das neue FAPM *SSynchronize* ein, das wir zur Modellierung der unterschiedlichen Sequenznummern einer PDU im Kanaltyp `"Active"` erstellt haben:

```
PROCESS SSynchronize    Uebertragung einer Dateneinheit mit unterschiedlichen Sequenznummern
                      ( sigd : Any CONSTANT ;   Signalisierungsinformation
                        tmax : Nat;             Max. Anzahl an Wiederholungen
                        ir : Nat )              Anzahl an Sequenznummern
IMPORT Constants;
BODY
  VARIABLES
    sobuf : SUBSET( fkey × sigd);   Signalisierungsinformationsspeicher
    smin : fkey;                    Sequenznummer des ersten Signals
    cRq : fkey;                     Sequenznummer des naechsten Signals
  INIT ≙ sobuf = ∅ ∧ smin = 0 ∧ cRq = 0;
  ACTIONS
    SRq ( krq : fkey; s : sigd ) ≙ Uebergabe von Signalisierungsinfo. durch den Dienstn.
      ∀ t ∈ sigd :: (smin, t) ∉ sobuf ∧
      sobuf′ = sobuf ∪ {(k, s) | between(smin,k,(smin + tmax) mod ir)} ∧
```

[10] In der Realität wird die Auslösung von Wiederholungen durch Timer gesteuert.

```
    smin' = smin ∧ cRq' = cRq;
  SuRq ( krq : fkey; s : sigd ) ≜
    Uebergabe von Signalisierungsinformation an die Protokollsubinstanz S
    (krq, s) ∈ sobuf ∧
    cRq' = inc(cRq) ∧
    sobuf' = sobuf \ { (k, s) | k = (cRq + 1 - is) mod ir } ∧
    smin' = smin;
  SCf ( krq ) ≜ Implizite Bestaetigung der Auslieferung von Signalisierungsinformation an den Dienstn.
    inc(krq) = cRq ∧
    sobuf' = sobuf \ { (k, s) | between(cRq, k, (cRq + tmax) mod ir) } ∧
    smin' = cRq ∧ cRq' = cRq;
  SuCf ( krq ) ≜ Implizite Bestaetigung der Auslieferung veralteter Signalisierungsinformation
    inc(krq) ≠ cRq ∧
    sobuf' = sobuf ∧ smin' = smin ∧ cRq' = cRq;
END
```

In XTP wird die PDU, mit der der Wunsch für einen neuen Datentransfermodus zum Partner gesendet wird, durch einen eindeutigen Wert in Feld `sync` gekennzeichnet. In der Bestätigungs-PDU ist derselbe Wert im Feld `echo` enthalten. Die Bestätigung wird vom Initiator des Dienstes nur akzeptiert, wenn der Wert in `echo` der zuletzt gesendeten Sequenznummer in `sync` entspricht. Bei einer erneuten Übertragung einer PDU zur Umschaltung des Datentransfermodus wird, um die Bestätigungen unterscheiden zu können, die nächsthöhere Sequenznummer zugewiesen. In unserer Lösung nutzen wir aus, daß zu jedem Zeitpunkt immer nur ein Dienstelement unbestätigt sein darf. Damit können wir den durch *SBufferKey* und *RBufferKey* spezifizierten Protokollmechanismus unverändert übernehmen. Wir setzen lediglich an der Schnittstelle zum Dienstnehmer einen durch das FAPM *SSynchronize* modellierten zusätzlichen Protokollmechanismus ein, der ein zu sendendes Signal zwischenspeichert und es (eventuell mehrfach) zur Übertragung an das FAPM *SBufferKey* übergibt. Dadurch erhält jede Wiederholung des Signals in *SBufferKey* eine eigene Sequenznummer.

In FAPM *SSynchronize* nennen wir die Sendeaktion an der Schnittstelle zum Dienstnehmer ebenfalls `SRq`. Zur Unterscheidung erhält die Sendeaktion an der Schnittstelle zu FAPM *SBufferKey* den Bezeichner `SuRq`. Implizite Bestätigungen, die durch das FAPM *SShowDelivery* modelliert werden, werden nur teilweise an den Dienstnehmer weitergereicht. Die weitergereichten Bestätigungen werden durch die Aktion `SCf` und die nicht weitergereichten durch `SuCf` beschrieben. Der Zwischenspeicher in *SSynchronize* ist `sobuf`. Die Sequenznummervergabe wird durch die Zähler `smin` und `cRq` modelliert. `smin` bezeichnet bei der Datenübergabe vom Dienstnehmer die Sequenznummer der ersten Übertragung. `cRq` entspricht dem Zähler in FAPM *SBufferKey*. Signalisierungsinformation wird vom Dienstnehmer übergeben (Aktion `SRq`) und höchstens `tmax`-mal an den durch FAPM *SBufferKey* modellierten Protokollmechanismus weitergereicht (Aktion `SuRq`). Die Wiederholung ist beendet, wenn die bei der letzten Sendung verwendete Sequenznummer bestätigt wird (Aktion `SCf`). Bestätigungen bereits wiederholter Signale werden ignoriert (Aktion `SuCf`). SCs, die durch *SSynchronize* und weitere FAPMs realisiert sind, werden in Abschnitt 6.3.3 vorgestellt. Der Beweis, daß diese SCs von der Protokollspezifikation erbracht werden, ist sehr einfach, da er auf die Theoreme des Frameworks zurückgeführt werden kann.

Bevor wir auf die Instantiierung der einzelnen FAPMs eingehen, beschreiben wir zunächst wieder die Parameterersetzungen, die in (fast) allen FAPMs verwendet werden:

```
pdu ≜ XTPpdu;
pci ≜ XTPpdu;
spci : SET[ XTPpdu → XTPpdu] ≜ [ x ∈ XTPpdu ↦ x ];
SigdTypes = { "FirstDuplex", "FirstSimplex", "NewQ", "WClose",
              "Final", "UAbort" };
sigd ≜ [ sTYPE : SigdTypes, sNOERR : {0, 1}, sNOCHECK : {0, 1},
         sNOFLOW : {0, 1}, sRES : {0, 1}, sFASTNAK : {0, 1},
         sSEQ : DWord, sDESTADR : { x | x ∈ Nat ∧ x < 2^(2^19 − 2^5 − 1)} ];
usd ≜ sigd;
fkey ≜ Word ∪ {"notsent"};
fpkey ≜ Word ∪ {"notsent"};
mseg = 1;
skk ≜ [ x ∈ Word ∪ {"notsent"} ↦ x ];
skn ≜ SET[ Word ∪ {"notsent"} ↦ 0 ];
skm ≜ SET[ Word ∪ {"notsent"} ↦ 1 ];
usdsize ≜ SET[ usd ↦ 1 ];
usdsplit : SET[ (Nat × usd) → usd] ≜ [ (n,d) ∈ (Nat × usd) ↦ d ];
encpdu ≜ { p ∈ XTPpdu | ( ( cmdNOCHECK(p.h.cmd) = 0 ∧
                            p.h.check = xsum(p.h.dlen + 32, p) ) ∨
                          ( cmdNOCHECK(p.h.cmd) = 1 ∧
                            p.h.check = xsum(32, p.h) ) ) };
ir ≜ 2^32;
```

Die Prozeßtypparameter der FAPMs für die Signalisierungskanäle werden mit Ausnahme der Parameter zur Definition der Nutzdaten und der Anzahl an Sequenznummern durch dieselben Werte ersetzt wie die Parameter der FAPMs für den Datentransfers. Als Nutzdaten werden Signale des Datentyps `sigd` übertragen, die Information über den realisierten Dienst enthalten (Feld `sTYPE`). Außerdem ist der vom Dienstnehmer vorgesehene Datentransfermodus als Dienstgüteparameter enthalten, der durch die Belegung der fünf Steuerflags `sNOERR`, `sNOCHECK`, `sNOFLOW`, `sRES` und `sFASTNAK` angezeigt wird. Das Feld `sSEQ` enthält die Sequenznummer der als letztes gesendeten Dateneinheit. Diese Information ist für den ordnungsgemäßen Abbau der Datenkanäle notwendig. Die Zieladresse der Partnerstation ist im Feld `sDESTADR` enthalten. Die Signalisierungsinformation wird durch Sequenznummern mit einer Länge von vier Bytes identifiziert (`ir`$\triangleq 2^{32}$).

In den FAPMs *SBufferKey* und *RBufferKey* ersetzen wir den Parameter `skey`, der die übertragenen Sequenznummern enthält, durch

```
skey ≜ [ p ∈ XTPpdu ↦
           CASE ChType = "First" ∧ cmdPFORMAT(p.h.cmd) = FIRST → { 0 },
                ChType = "Active" ∧ cmdEND(p.h.cmd) = 0 ∧
                  cmdSREQ(p.h.cmd) = 1 ∧
                  ( cmdWCLOSE(p.h.cmd) = 1 ∨
                    cmdPFORMAT(p.h.cmd) ∈ {FIRST,TCNTL} ) → { p.h.sync },
                ChType = "Last" ∧ cmdEND(p.h.cmd) = 1 ∧
                  cmdPFORMAT(p.h.cmd) ≠ DIAG → { 0 }
                OTHERWISE → "≪≫" ];
```

`ChType` ist der neue Prozeßtypparameter, der den verwendeten Kanaltyp bezeichnet. Die Kanaltypen `"First"` und `"Last"` werden im Verlauf einer Verbindung jeweils nur zur Übertragung von PDUs für ein einziges Signal verwendet, da sowohl der Verbindungsaufbau als

auch der endgültige Abbau implizit erfolgt. Damit wird im Normalfall jeweils auch nur eine PDU über "First" und "Last" gesendet, die jedoch bei Übertragungsfehlern wiederholt wird. Die Sequenznummer dieser PDU ist nach den durch die FAPMs modellierten Mechanismen daher immer 0. In Kanaltyp "First" enthalten die Verbindungsaufbau-PDUs des PDU-Formats FIRST folglich die Sequenznummer 0 und alle anderen PDUs "<<>>", da sie keine Signalisierungsinformation für den Verbindungsaufbau enthalten. Entsprechend ist 0 in Kanaltyp "Last" nur in PDUs enthalten, deren Steuerflag END das Verbindungsende anzeigt. Ferner darf die PDU nicht vom Format DIAG sein, das einen vom Diensterbringer initiierten Verbindungsabbruch anzeigen würde. In Kanaltyp "Active" zeigt der Wert aus Feld sync die Sequenznummer der PDU an. PDUs dienen zur Aushandlung neuer Datenraten, wenn sie zum einen das Format FIRST oder TCNTL besitzen und zum anderen das Steuerflag SREQ, aber nicht END, gesetzt ist. In PDUs zur Abbauanzeige eines Datenkanals ist neben SREQ auch das Steuerflag WCLOSE, jedoch nicht das Flag END gesetzt. In allen Kanaltypen wird der Parameter mb mit dem Wert 1 instantiiert, weil eine PDU nur ein über den Signalisierungskanal übertragenes Signal enthält. Ferner werden rcc durch 0 und rcu durch 1 ersetzt, da jeder Dienst entweder nur einmal ausgeführt wird, oder erst nach seiner Bestätigung wiederholt werden darf.

Die Darstellung der in den XTP-PDUs enthaltenen Signalisierungsdaten ist etwas komplexer als die der Nutzdaten, da sie nicht in bestimmten Segmenten einer PDU zusammengefaßt sind, sondern durch bestimmte Kombinationen von PDU-Formaten und Steuerflagbelegungen beschrieben werden. Der Parameter susd in den FAPMs *SBufferUsd* und *RBufferUsd*, der die Signalisierungsinformation referenziert, wird durch

```
susd ≙ LET q (p : XTPpdu; t : SigdTypes) ≙
            [ sTYPE ↦ t, sNOERR ↦ cmdNOERR(p.h.cmd),
              sNOCHECK ↦ cmdNOCHECK(p.h.cmd),
              sNOFLOW ↦ cmdNOFLOW(p.h.cmd),
              sRES ↦ cmdRES(p.h.cmd), sFASTNAK ↦ cmdFASTNAK(p.h.cmd),
              sSEQ ↦ CASE cmdPFORMAT(p.h.cmd) = FIRST →
                            p.h.dlen - p.p.is.alen - p.p.is.tlen,
                          cmdPFORMAT(p.h.cmd) = DATA →
                            (p.h.seq + p.h.dlen) mod 2^64,
                          OTHERWISE → p.h.seq ]
      IN [ (p,k) ∈ (XTPpdu × Word) ↦
             CASE ChType = "First" ∧ k = 0 →
                    IF (cmdRCLOSE(p.h.cmd) = 0)
                    THEN q(p,"FirstDuplex") ELSE q(p,"FirstSimplex"),
                  ChType = "Active" ∧ k = p.h.sync →
                    IF (cmdWCLOSE(p.h.cmd) = 0)
                    THEN q(p,"NewQ") ELSE q(p,"WClose"),
                  ChType = "Last" ∧ k = 0 →
                    IF (cmdWCLOSE(p.h.cmd) = 0)
                    THEN q(p,"UAbort") ELSE q(p,"Final") ];
```

ersetzt. Die Hilfsfunktion q(p,t) beschreibt den Aufbau eines Signals von Typ sigd in der PDU p, das zur Realisierung des Dienstes der Art t übertragen wird. Die Flags sNOERR, sNOCHECK, sNOFLOW, sRES und sFASTNAK in diesem Signal sigd entsprechen den gleichnamigen Steuerflags in der PDU. Die Anzeige sSEQ enthält immer die Sequenznummer des zuletzt gesendeten Nutzdatenbytes. Der Diensttyp, der im Feld sTYPE enthalten ist, hängt zum einen

vom modellierten Kanaltyp und zum anderen von den Steuerflags `RCLOSE` und `WCLOSE` ab.

Die Parameter `sackm` und `sacko` in den FAPMs *SAcknowledge*, *RAcknowledge* und *SShow-Delivery* modellieren innerhalb einer Bestätigungs-PDU die Sequenznummern der von der Empfängerinstanz bestätigten Signale (siehe auch Abschnitt 6.2.2) und werden folgendermaßen ersetzt:

```
sackm ≙ [ p ∈ XTPpdu ↦
            CASE ChType = "First" → 0,
                 ChType = "Active" ∧ cmdRCLOSE(p.h.cmd) = 1 ∧
                   cmdPFORMAT(p.h.cmd) = CNTL → p.p.cs.cc.echo,
                 ChType = "Active" ∧ cmdRCLOSE(p.h.cmd) = 1 ∧
                   cmdPFORMAT(p.h.cmd) = ECNTL → p.p.cs.ec.echo,
                 ChType = "Active" ∧
                   ( cmdRCLOSE(p.h.cmd) = 1 ∨ cmdSREQ(p.h.cmd) = 0 ) ∧
                   cmdPFORMAT(p.h.cmd) = TCNTL → p.p.cs.tc.echo,
                 OTHERWISE → "≪≫" ];

sacko : SET[XTPpdu → (SUBSET(fpkey) ∪ {"≪≫"})] ≙
  [ p ∈ XTPpdu ↦
      IF ( ChType = "First" ∨
           ( ChType = "Active" ∧
             ( cmdRCLOSE(p.h.cmd) = 1 ∨ cmdSREQ(p.h.cmd) = 0 ) ) )
      THEN ∅ ELSE "≪≫" ];
```

Im Kanaltyp `"First"`, der die Verbindungsaufbau-Dienste unterstützt, gilt jede PDU, die der Verbindungsaufbauinitiator von seinem Partner erhält, als Bestätigung. Weil der Verbindungsaufbau-PDU jedoch immer die Sequenznummer `0` zugeordnet ist, ersetzen wir `sackm` durch `0`. Im Kanaltyp `"Active"` enthält das Feld `echo` die Kennung bestätigter Dateneinheiten. Eine Neuaushandlung des Datentransfermodus wird nur durch eine PDU des Formats `TCNTL` bestätigt, in der das Steuerflag `SREQ` nicht gesetzt ist. In Bestätigungen von Datenkanalschließungen, die in PDUs der Formate `CNTL`, `ECNTL` und `TCNTL` erfolgen, ist hingegen das Flag `RCLOSE` gesetzt. Im Kanaltyp `"Last"` sind keine Bestätigungen vorgesehen, so daß `sackm` immer der Wert `"<<>>"` zugeordnet wird. Analog zu `sacko` referenziert der Parameter `sackm` entweder die leere Menge $\emptyset$ oder den Wert `"<<>>"`.

Explizite Zurückweisungen gibt es nur in Kanaltyp `"Last"`, in dem jede Nachricht vom Verbindungspartner als Hinweis auf einen Fehler gewertet wird, der eine Wiederholung der Dienst-PDU zuläßt. Der Parameter `snak` wird deshalb durch

```
snak ≙ [ p ∈ XTPpdu ↦ IF (ChType = "Last" ∧ cmdEND(p.h.cmd) = 0)
                       THEN {0} ELSE "≪≫" ];
```

ersetzt. Der Parameter `ma` erhält den Wert 0.

Weitere FAPMs sind *RDuplicates*, *RReorderings*, *RPhantoms* und *SSendInOrder*, in denen wir folgende Parameterersetzungen durchführen:

```
tr ≙ 0;
td ≙ 0;
pd ≙ SUBSET(sigd);
```

FAPMs zur Beschreibung von Liveness-Eigenschaften werden für die Signalisierungskanäle `"First"`, `"Active"` und `"Last"` nicht definiert, da ständig verlorengegangene Verbindungsverwaltungs-PDUs immer zu einem Verbindungsabbruch durch den Diensterbringer führen.

Deshalb werden für die Dienste der Verbindungsverwaltung, die vom Dienstnehmer initiiert sind, keine Liveness-FAPMs modelliert.

Als nächstes betrachten wir die FAPMs, die den Kanal spezifizieren, über den vom Diensterbringer initiierte Verbindungsabbruchnachrichten (Provider-Aborts) übertragen werden. Da diese Verbindungsabbruchmeldungen keine Nutzdaten enthalten und auch keine Bestätigungs- bzw. Kreditmechanismen notwendig sind, genügen die Safety-FAPMs *SBufferKey*, *RBufferKey*, *SReject* und *RReject* zur Beschreibung des entsprechenden Kanals. Wie in Abschnitt 5.2.3 erläutert wurde, benennen wir die Aktion `Rq` in `PAbI` um, da sie nicht zur Sendung von Daten, sondern zur Auslieferung einer Verbindungsabbruchanzeige dient, die von der Protokollinstanz an diesem Dienstzugangspunkt initiiert wurde. Analog dazu wird die Aktion `In` in `PAbP` umbenannt und beschreibt Verbindungsabbruchmitteilungen, die vom Verbindungspartner initiiert wurden. Die Safety-FAPMs werden mit den folgenden Parametrisierungen verwendet:

```
skey ≙ [ p ∈ XTPpdu ↦ IF ( cmdPFORMAT(p.h.cmd) = DIAG ∧
                                cmdEND(p.h.cmd) = 1 )
                           THEN {0} ELSE "≪ ≫" ];

snak ≙ [ p ∈ XTPpdu ↦ IF ( cmdEND(p.h.cmd) = 0 )
                           THEN {0} ELSE "≪ ≫" ];
```

Im Gegensatz zu den anderen Kanälen der Verbindungsverwaltung werden hier auch Liveness-FAPMs instantiiert. Es muß sichergestellt sein, daß der Partner eines Dienstnehmers, dem ein Abbruch signalisiert wird (Aktion `PAbI`), ebenfalls eine Verbindungsabbruchnachricht erhält (Aktion `PAbP`). Wie bei den Kanälen des Datentransfers setzen wir die Liveness-FAPMs *SLiveMRq*, *RLiveARq* und *RLiveIn* ein. Wir verwenden dazu die folgenden Parametererersetzungen:

```
sackm ≙ [ p ∈ XTPpdu ↦ "≪ ≫" ];
sacko ≙ [ p ∈ XTPpdu ↦ "≪ ≫" ];
scre ≙ [ p ∈ XTPpdu ↦ "≪ ≫" ];
stack ≙ [ p ∈ XTPpdu ↦ "on" ];
stcre ≙ [ p ∈ XTPpdu ↦ "on" ];
mcr ≙ 0;
rcc ≙ 0;
rcu ≙ 1;
ma ≙ 0;
mo ≙ 0;
mr ≙ 0;
tg ≙ "infty";
```

6.2.5 Komposition der FAPMs der Signalisierungskanäle

Aus den oben beschriebenen FAPMs werden die Spezifikationen der vier Signalisierungskanaltypen durch cTLA$^+$-Kombination erzeugt (siehe Abschnitt 4.3.2). Auf der linken Seite der Tabelle 6.4 sind die FAPMs angegeben, aus denen die Spezifikationen der Kanaltypen `"First"`, `"Active"` und `"Last"` komponiert werden. Die verwendeten FAPMs spiegeln wieder, daß Fehlerbehebung in `"First"` und `"Active"` durch positive Bestätigungen erfolgt,

FAPM	"First"	"Active"	"Last"
SBufferKey	⋆	⋆	⋆
RBufferKey	⋆	⋆	⋆
SBufferUsd	⋆	⋆	⋆
RBufferUsd	⋆	⋆	⋆
SAcknowledge	⋆	⋆	
RAcknowledge	⋆	⋆	
SReject			⋆
RReject			⋆
RDuplicates	⋆	⋆	⋆
RReorderings	⋆	⋆	⋆
RPhantoms	⋆	⋆	⋆
SSendInOrder	⋆	⋆	⋆
SShowDelivery		⋆	
SSynchronize		⋆	

FAPM	Prov.-Ab.
SBufferKey	⋆
RBufferKey	⋆
SReject	⋆
RReject	⋆
SLiveMRq	⋆
RLiveARq	⋆
RLiveIn	⋆

Tabelle 6.4: Komposition der FAPMs der Signalisierungsdatenkanäle

während man in "Last" explizite Zurückweisungen einsetzt (siehe Abschnitt 2.2.2). "Active" unterscheidet sich von "First" durch ein zusätzliches FAPM für Empfangsbestätigungen und das in Abschnitt 6.2.4 erläuterte FAPM *SSynchronize*, durch das die Sequenznummervergabe modifiziert wird.

Die Zusammensetzung des Kanaltyps für Provider-Aborts wird in der Tabelle 6.4 auf der rechten Seite angegeben.

6.2.6 Instantiierung der FAPMs der Intraconnection-Coordination

Aufgabe der FAPMs dieser Gruppe ist die Koordinierung der 36 Datentransferkanäle und der acht Signalisierungsdatenkanäle. In XTP ist dabei vor allem die Verwaltung der Datentransferkanäle von Interesse. Zwar darf innerhalb einer Verbindung zu jedem Zeitpunkt in jede Richtung nur ein Datenkanal geöffnet sein, jedoch kann bei der Änderung des Datentransfermodus ein offener Kanal im Laufe der Verbindung geschlossen und dafür ein anderer Kanal geöffnet werden.

Beim Verbindungsaufbau wird nur ein Simplexdatenkanal (Diensttyp "FirstSimplex") oder in beiden Senderichtungen je ein Simplexdatenkanal (Diensttyp "FirstDuplex") geöffnet. Bei der Neuaushandlung des Datentransfermodus (Diensttyp "NewQ") wird der offene Kanal zwischen dem Dienstinitiator und seinem Partner geschlossen und dafür ein neuer Kanal geöffnet, der den zukünftigen Transfermodus beschreibt. Dabei erfolgt die Kanalschließung sofort bei Initiierung des Dienstes, die Eröffnung des neuen Kanals jedoch erst bei Eintreffen der Dienstbestätigung. Im Normalfall werden die beiden Simplexkanäle vor dem Verbindungsende durch die Sendeinstanzen geschlossen (Diensttyp "WClose"). Bei einem Verbindungsabbruch (Datentypen "Final", "UAbort" und Provider-Abort) werden die Kanäle hingegen während des Verbindungsabbruchs geschlossen.

Wir beschreiben zunächst wieder Parameterersetzungen, die alle FAPMs betreffen:

```
SigdTypes = { "FirstDuplex", "FirstSimplex", "NewQ", "WClose",
              "Final", "UAbort" };
sigd ≜ [ sTYPE : SigdTypes, sNOERR : {0, 1}, sNOCHECK : {0, 1},
         sNOFLOW : {0, 1}, sRES : {0, 1}, sFASTNAK : {0, 1},
         sSEQ : DWord, sDESTADR : { x | x ∈ Nat ∧ x < 2^(2^19 − 2^5 − 1)} ];
sHStot ≜ [ s ∈ sigd ↦ 1 ];
sHSno ≜ [ s ∈ sigd ↦ 1 ];
```

Die Menge der Signale wird durch den Datentyp `sigd` beschrieben (siehe Abschnitt 6.2.4). Da alle Dienste der Verbindungsverwaltung entweder unbestätigt oder implizit bestätigt sind (siehe Tabelle 6.3), gibt es keinen Handshake, und jeder Dienst wird durch genau ein Signal realisiert. Deshalb haben die Parameter `sHStot`, der die Anzahl der zur Erbringung eines Dienstes auszutauschenden Signale angibt, und `sHSno`, der die Nummer der aktuellen Signalisierungsdateneinheit anzeigt, jeweils den Wert 1.

Die Koordinierung der Signalisierungskanäle wird durch das FAPM *ConnectionState* spezifiziert:

```
sfinal ≜ [ s ∈ sigd ↦ IF (s.sTYPE ∈ {"Final", "UAbort"}) THEN 1 ELSE 0 ];
```

Der Parameter `sfinal` zeigt an, ob das Signal einen vom Dienstnehmer initiierten Verbindungsabbruch beschreibt. Eine Verbindung wird nur durch die Dienste `"Final"` oder `"UAbort"` vom Dienstnehmer abgebrochen. Da `sfinal` immer die Nummer des Signals in einem Handshake angibt, durch das die Verbindung abgebaut werden soll, wird in diesem Fall die Nummer 1 und sonst 0 referenziert. Der Parameter `beginSt` im FAPM zeigt an, ob die Station, der das FAPM zugeordnet ist, eine Verbindung eröffnen darf. Da XTP keinen überlagerten Verbindungsaufbau (Contention) kennt[11], wird das Recht des Verbindungsaufbaus nur einer der beiden Stationen gegeben.

Die Koordinierung der Datentransferkanäle wird durch die FAPMs *DataChanOpenW*, *DataChanOpenR*, *SMarkFinal*, *RMarkFinal*, *RDeliverBuff* und *RStopOld* modelliert, in denen die folgenden Parameterinstantiierungen vorgenommen werden:

```
sOpenIW ≜ [ s ∈ sigd ↦ IF (s.sTYPE ∈ {"FirstDuplex", "FirstSimplex", "NewQ"})
                       THEN 1 ELSE 0 ];
sOpenIR ≜ [ s ∈ sigd ↦ IF (s.sTYPE = "FirstDuplex") THEN 1 ELSE 0 ];
sOpenPW ≜ [ s ∈ sigd ↦ IF (s.sTYPE = "FirstDuplex" THEN 1 ELSE 0 ];
sOpenPR ≜ [ s ∈ sigd ↦ IF (s.sTYPE ∈ {"FirstDuplex", "FirstSimplex", "NewQ"})
                       THEN 1 ELSE 0 ];
sCloseIW ≜ [ s ∈ sigd ↦ IF (s.sTYPE ∈ {"NewQ", "WClose", "Final", "UAbort"})
                        THEN 1 ELSE 0 ];
sCloseIR ≜ [ s ∈ sigd ↦ IF (s.sTYPE ∈ {"Final", "UAbort"}) THEN 1 ELSE 0 ];
sClosePW ≜ [ s ∈ sigd ↦ IF (s.sTYPE ∈ {"Final", "UAbort"}) THEN 1 ELSE 0 ];
sClosePR ≜ [ s ∈ sigd ↦ IF (s.sTYPE ∈ {"NewQ", "WClose","Final", "UAbort"})
                        THEN 1 ELSE 0 ];
sOpenmode ≜ [ s ∈ sigd ↦
                IF (s.sTYPE = "NewQ") THEN "delayed" ELSE "immediate" ];
sClosemode ≜ [ s ∈ sigd ↦ "immediate" ];
```

[11] Wenn beide Stationen gleichzeitig versuchen, eine Verbindung aufzubauen, führt das zu zwei voneinander unabhängigen Verbindungen.

```
Qual ≜ SUBSET([ sNOERR : {0, 1}, sNOCHECK : {0, 1},
                sNOFLOW : {0, 1}, sRES : {0, 1}, sFASTNAK : {0, 1} ]);
sQualWold ≜ [ s ∈ sigd ↦
                IF (s.sTYPE = "NewQ")
                THEN Qual \ {[ sNOERR ↦ s.sNOERR, sNOCHECK ↦ s.sNOCHECK,
                               sNOFLOW ↦ s.sNOFLOW, sRES ↦ s.sRES,
                               sFASTNAK ↦ s.sFASTNAK ]}
                ELSE Qual ];
sQualRold ≜ [ s ∈ sigd ↦ Qual ];
sQualWnew ≜ [ s ∈ sigd ↦ {[ sNOERR ↦ s.sNOERR, sNOCHECK ↦ s.sNOCHECK,
                             sNOFLOW ↦ s.sNOFLOW, sRES ↦ s.sRES,
                             sFASTNAK ↦ s.sFASTNAK ]} ];
sQualRnew ≜ [ s ∈ sigd ↦ {[ sNOERR ↦ s.sNOERR, sNOCHECK ↦ s.sNOCHECK,
                             sNOFLOW ↦ s.sNOFLOW, sRES ↦ s.sRES,
                             sFASTNAK ↦ s.sFASTNAK ]} ];

sClosemodeW ≜ [ s ∈ sigd ↦
                  IF (s.sTYPE = "UAbort") THEN "immediate" ELSE "delayed" ];
sClosecRq ≜ [ s ∈ sigd ↦ { s.sSEQ } ];
```

Wie `sfinal` zeigen die Parameter `sOpenIW`, `sOpenIR`, `sOpenPW`, `sOpenPR`, `sCloseIW`, `sCloseIR`, `sClosePW` und `sClosePR` an, ob durch die PDU ein bestimmter Datenkanal geöffnet oder geschlossen wird (Wert 1) oder nicht (Wert 0). Durch `sOpenIW` wird beschrieben, ob der Dienstinitiator auf einem Datentransferkanal anfangen darf zu senden. Das ist in XTP der Fall, wenn entweder die Verbindung neu aufgebaut wird (Diensttypen `"FirstDuplex"` und `"FirstSimplex"`) oder im Rahmen einer Neuaushandlung des Transfermodus der Kanal eröffnet wird, der den neuen Modus anbietet (Diensttyp `"NewQ"`). Eine Empfangsberechtigung an den Dienstinitiator (`sOpenIR`) muß nur beim Aufbau einer neuen Duplex-Verbindung vergeben werden. Komplementär dazu erfolgt die Vergabe der Leserechte (`sOpenPR`) bzw. Schreibrechte (`sOpenPW`) beim Partner des Dienstinitiators. Eine Station kann einen Datentransferkanal zu ihrem Partner schließen (`sCloseIW`), wenn sie einen neuen Datentransfermodus aushandelt (`"NewQ"`), die Datenübertragung zum Partner endgültig schließt (`"WClose"`) oder die Verbindung beendet (`"Final"` und `"UAbort"`). Die eigene Empfangsberechtigung (`sCloseIR`) darf sie nur beim Verbindungsabbau schließen. Die Reaktion des Dienstempfängers (`sClosePR` oder `sClosePW`) verläuft dazu wiederum komplementär. Weitere Parameter sind `sOpenmode` und `sClosemode`. Sie zeigen an, ob die Sendeberechtigung auf einem Kanal vom Initiator zum Partner sofort nach Initiierung oder erst nach Bestätigung des Dienstes gegeben bzw. aufgehoben wird. Bei Datentransfermodusänderungen erfolgt die Vergabe der Sendeberechtigung erst nach der Bestätigung, ansonsten sofort.

Durch die im folgenden beschriebenen Parameter wird festgelegt, welche Datentransferkanäle geöffnet bzw. geschlossen werden. Das Feld `Qual` enthält die Menge aller erlaubten Datentransfermodi, die in der XTP-Spezifikation durch die Flags `NOERR`, `NOCHECK`, `NOFLOW`, `RES` und `FASTNAK` beschrieben werden. Die Parameter `sQualWold` und `sQualRold` geben an, welche Kanäle geschlossen werden sollen. Da in `sigd` keine Daten über den zu schließenden Kanal gespeichert sind, aber zu jedem Zeitpunkt immer nur höchstens ein Kanal offen ist, modellieren wir, daß alle Datenkanäle geschlossen werden. Von der Schließung ausgenommen bleibt bei Änderungen des Datentransfermodus nur der Kanal, der den neuen Datentransfermodus spezifiziert. Dieser Kanal wird geöffnet. Sein Bezeichner ist deshalb das einzige

Element der Mengenparameter `sQualWnew` und `sQualRnew`.

Der Parameter `sClosemodeW` zeigt an, ob eine Verbindungsabbauanzeige erst nach Auslieferung aller Dateneinheiten der betroffenen Kanäle erfolgen darf. Das ist in XTP immer der Fall, wenn es sich nicht um einen Verbindungsabbruch (`"UAbort"`) handelt. Ein weiterer Parameter `sClosecRq` referenziert die Sequenznummer der über den Kanal zuletzt gesendeten Dateneinheit, die dem Feld `sSEQ` der Signalisierungsinformation entspricht.

Je eine Kopie der FAPMs *DataChanOpenW*, *DataChanOpenR*, *SMarkFinal*, *RMarkFinal*, *RDeliverBuff* und *RStopOld* ist jeder der 36 Spezifikationen der Datentransferkanäle zugeordnet. Die Prozeßtypparameter `skk`, `rcc`, `rcu` und `tg` werden durch die gleichen Werte ersetzt wie in den FAPMs des Datentransfers der Kanalspezifikation (siehe Abschnitt 6.2.2). Der Parameter `lq` zeigt den Modus des Datentransferkanals an. Wir ersetzen ihn deshalb durch ein Tupel, das die Parameter `NOERR`, `NOCHECK`, `NOFLOW`, `RES` und `FASTNAK` enthält.

Das von den Dienstnehmern geforderte Verhalten bei der Initiierung von Diensten wird durch das FAPM *UserBehaviour* modelliert. Die Parameter dieses FAPMs stellen den endlichen Automaten dar, der das Dienstnehmerverhalten abbildet (siehe Abbildung 6.3). Die Parameter von *UserBehaviour* werden auf die folgende Weise ersetzt:

```
States ≙ { k | k ∈ Nat : k ≤ 8 } ∪ {"I","T"};
InitialStates ≙ { "I" };

TransdSRq : SUBSET (States × SigdTypes × States) ≙
  { ( "I", "FirstDuplex", 0 ), ( "I", "FirstSimplex", 1 ), ( 0, "NewQ", 3 ),
    ( 1, "NewQ", 4 ), ( 0, "WClose", 5 ), ( 1, "WClose", 8 ),
    ( 1, "Final", "T" ), ( 7, "Final", "T" ) } ∪
  { ( x, "UAbort", "T" ) | x ∈ { k | k ∈ Nat : k ≤ 8 } };

TransdSIn : SUBSET (States × SigdTypes × States) ≙
  { ( "I", "FirstDuplex", 0 ), ( "I", "FirstSimplex", 2 ),
    ( 0, "WClose", 1 ), ( 2, "WClose", 7 ), ( 3, "WClose", 4 ),
    ( 5, "WClose", 6 ) } ∪
  { ( x, "Final", "T" ) | x ∈ { k | k ∈ Nat : k ≤ 8 } } ∪
  { ( x, "UAbort", "T" ) | x ∈ { k | k ∈ Nat : k ≤ 8 } };

TransdSCf : SUBSET (States × States) ≙
  { ( 3, 0 ), ( 4, 1 ), ( 5, 2 ), ( 6, 7 ), ( 8, 7 ) };
```

`States` gibt den Zustandsbereich und `InitialStates` den Initialzustand des Automaten an. Durch `TransdSRq`, `TransdSIn` und `TransdSCf` werden die Zustandsübergänge beim Schalten der Aktionen `SRq`, `SIn` bzw. `SCf` beschrieben.

SLivePAbI, das einzige Liveness-FAPM der Intraconnection-Coordination, beschreibt, daß eine Verbindung garantiert abgebrochen wird, wenn ein bestimmter Dienst nicht seinem Empfänger angezeigt werden kann. Die Ersetzung seiner Parameter entspricht den in Abschnitt 6.2.4 vorgegebenen Regeln.

6.2.7 Komposition der FAPMs der Intraconnection-Coordination

Wie oben bereits gesagt wurde, werden den beiden Dienstzugangspunkten einer Verbindung jeweils 36 Kopien der FAPMs *DataChanOpenW*, *DataChanOpenR*, *SMarkFinal*, *RMarkFinal*, *RDeliverBuff* und *RStopOld* zugeordnet. Die FAPMs *DataChanOpenW* und *SMarkFinal* werden mit den FAPMs der 18 Datenkanäle gekoppelt, bei denen der Dienstzugangspunkt

als Sender fungiert. Die FAPMs *DataChanOpenR*, *RMarkFinal*, *RDeliverBuff* und *RStopOld* sind dagegen mit den Kanälen gekoppelt, für die der Dienstzugangspunkt die Rolle des Empfängers besitzt. Von den FAPMs *ConnectionState*, *UserBehaviour* und *SLivePAbI* wird jedem Dienstzugangspunkt nur eine Kopie zugeordnet. Mit Ausnahme von `beginSt` in FAPM *OpenConnection* werden alle FAPMs in beiden Dienstzugangspunkten durch dieselben Werte ersetzt. `beginSt` wird an dem Dienstzugangspunkt, von dem aus die Verbindung aufgebaut werden darf, mit dem Wert `"Initiator"` instantiiert. Am anderen Dienstzugangspunkt wird er durch `"Partner"` ersetzt.

Die Komposition der FAPMs der Intraconnection-Coordination mit denen der Datentransferkanäle erfolgt durch Kopplung der Aktionen `Rq` und `In`. Dagegen komponiert man sie mit den Spezifikationen der Signalisierungskanäle, indem die Aktionen `SRq`, `SIn`, `PAbI` und `PAbP` mit den gleichnamigen Aktionen der Signalisierungskanal-Spezifikationen gekoppelt werden.

6.2.8 Instantiierung der FAPMs der Interconnection-Coordination

Die Koordinierung unterschiedlicher Verbindungen wird durch die FAPMs *OpenConnections* und *FindDestination* beschrieben, die die Begrenzung der in einer Station gleichzeitig geöffneten Verbindungen bzw. die Auswahl einer geeigneten Verbindung spezifizieren. Die FAPMs *AssignCNo*, *SelectCNo*, *FreeCNo* und *LiveFreeCNo* modellieren den Mechanismus zur Verwaltung von Verbindungskennungen.

Die FAPMs enthalten eindeutige Verbindungsnummern `no` aus der Menge der natürlichen Zahlen. Wir interpretieren `no` als Bitfolge, die die Stationsadressen des Verbindungsinitiators und seines Partners enthalten:

$$no \triangleq \text{Offset} \circ \text{Initiatoradresse} \circ \text{Partneradresse}$$

Die letzten $2^{19} - 2^5$ Bits einer Verbindungsnummer `no` beschreiben also immer die Adressen der an einer Verbindung beteiligten Stationen. XTP unterstützt verschiedene Adreßformate (z.B. IP-Adressen, ISO-CNLP-Adressen, Xerox-Netzwerkadressen) mit unterschiedlich großen Adreßräumen. Zur Vereinfachung ordnen wir den Adressen immer den größten in den XTP-PDUs darstellbaren Adreßraum von $2^{2^{19}-2^5-1}$ unterschiedlichen Werten zu.

Aufgrund der Interpretation von `no` führen wir im FAPM *OpenConnections* die folgende Parameterersetzung der Prozeßtypparameter durch:

$\texttt{cno} \triangleq \{\ \texttt{x} \in \texttt{Nat} \mid \texttt{x} < 2^{63}\ \};$
$\texttt{cona} \triangleq \{\ \texttt{x} \in \texttt{Nat} \mid (\texttt{x div } 2^{2^{19}-2^5-1})\ \texttt{mod } 2^{2^{19}-2^5-1} = \texttt{MYADR}\ \};$
$\texttt{conb} \triangleq \{\ \texttt{x} \in \texttt{Nat} \mid \texttt{x mod } 2^{2^{19}-2^5-1} = \texttt{MYADR}\ \};$

`cno` bezeichnet die Menge aller in einer Station eindeutigen Verbindungskennungen. Derartige lokal eindeutige Kennungen werden in den XTP-PDUs in den letzten 63 Bits des Feldes `key` übertragen. Daraus folgt, daß jede Station 2^{63} unterschiedliche Verbindungskennungen vergeben darf. Die Topologie des Rechnernetzes, dessen Transportfunktionen durch XTP realisiert werden, wird durch die Parameter `cona` und `conb` bestimmt. Wir unterstellen dabei, daß die Station, zu deren Spezifikation das FAPM zugeordnet ist, die lokale Adresse `MYADR` verwendet. Die Menge `cona` enthält die Nummern `no` aller Verbindungen, die von der Station `MYADR` aus aufgebaut werden dürfen. `conb` ist die Menge aller Verbindungsnummern, in denen `MYADR` der Partner des Verbindungsinitiators ist.

Im FAPM *FindDestination* werden die folgenden Parameterersetzungen durchgeführt:

```
con ≜ { x ∈ Nat | (x div 2^(2^19−2^5−1)) mod 2^(2^19−2^5−1) = MYADR };
adrlist : SET[ sadr → SUBSET( Nat )] ≜
                [ s ∈ sadr ↦ { x ∈ Nat | x mod 2^(2^19−2^5−1) = s } ];
sdest : SET[ sigd → sadr ] ≜ [ s ∈ sigd ↦ s.DESTADR ];
```

Durch `sdest` wird der Teil einer Signalisierungsdateneinheit spezifiziert, die die Adresse des Empfängers beschreibt. In unserer Transportschnittstellenbeschreibung entspricht das dem Feld `sDESTADR` des Datentyps `sigd` (siehe Abschnitt 6.2.4). `con` ist die Menge aller Verbindungsnummern, die von der Station `MYADR` aufgebaut werden dürfen. Sie hat somit die gleiche Wertbelegung wie der oben beschriebene Parameter `cona`. Durch `adrlist` wird ein Adreßverzeichnis beschrieben, das jeder möglichen Partneradresse die zum Partner führenden Verbindungen zuordnet. Jedes Feld `s` dieses Verzeichnises wird mit der Menge aller Verbindungsnummern instantiiert, deren Partneradreßbereich den Wert `s` besitzt. Da XTP unterschiedliche Adreßformate unterstützt, ist die Instantiierung des Parameters `sadr`, der die Menge aller Stationsadressen beschreibt, Aufgabe des Protokollimplementierers.

In den FAPMs zur Verwaltung der Verbindungskennungen müssen neben den oben sowie in den vorgehenden Abschnitten beschriebenen Parameterersetzungen nur die beiden Parameter `sTCN` und `sPCN` ersetzt werden:

```
sTCN : SET[pci → cno ∪ {"idle"}] ≜
  [ p ∈ pci ↦ IF (keyRTN(p.h.key) = 0) THEN keyKEY(p.h.key) ELSE "idle" ];
sPCN : SET[pci → cno ∪ {"idle"}] ≜
  [ p ∈ pci ↦ IF (keyRTN(p.h.key) = 1) THEN keyKEY(p.h.key) ELSE "idle" ];
```

`sTCN` beschreibt den Teil der PCI einer PDU, in der eine Verbindungskennung enthalten ist, die im Bereich des Senders der PDU eindeutig ist. Eine derartige Kennung befindet sich im Feld `key` des Kopfes einer XTP-PDU, wenn das dort ebenfalls enthaltene Return-Flag nicht gesetzt ist (siehe Abschnitt 6.1.3). Dagegen beschreibt `sPCN` die Referenz auf eine Verbindungskennung, die der Empfänger der PDU eindeutig zuordnen kann. Das Feld `key` enthält eine derartige Kennung, wenn das Return-Flag gesetzt ist.

6.2.9 Komposition der FAPMs der Interconnection-Coordination

Je zwei Kopien der FAPMs *AssignCNo*, *FreeCNo* und *LiveFreeCNo* werden mit jeder Verbindungsspezifikation gekoppelt. Dabei wird jedem Dienstzugangspunkt in der Verbindungsspezifikation ein Satz dieser FAPMs zugeordnet. Aus dieser erweiterten Verbindungsspezifikation wird durch die cTLA$^+$-Prozeßfeldoperation ein Feld an Verbindungsspezifikationen erzeugt.

Davon unabhängig werden die FAPMs *OpenConnections*, *FindDestination* und *SelectCNo* zu der Spezifikation der Verbindungsverwaltung eines Rechnernetzknotens komponiert. Daraus wird ebenfalls ein Prozeßfeld gebildet, das die Verbindungsverwaltung aller Knoten des Rechnernetzes modelliert.

In einem weiteren Schritt werden die beiden Prozeßfelder miteinander komponiert. Die daraus resultierende Spezifikation beschreibt das gesamte durch das Protokoll XTP erbrachte Transportsystem.

6.2.10 Instantiierung der AMCs

Die Basiskanäle zum Transport von Nutzdaten und Signalisierungsinformation dürfen alle übertragenen Daten verfälschen und auch Phantome ausliefern. Wir verlangen jedoch, daß

Verfälschungen und Phantome von der Empfängerinstanz durch ihre falsche Prüfsumme erkannt werden können. Außerdem sollen die Kanäle garantieren, daß sie nicht beliebig alte PDUs ausliefern. Um Aussagen über die Transportdienstqualität bei eingeschränkter Fehlerbehebung machen zu können, verlangen wir weiterhin, daß die relativen Häufigkeiten an Verfälschungen, Verlusten und Phantomen nicht bestimmte Grenzen überschreiten. Ferner müssen die Basiskanäle lebendig sein und, wenn regelmäßig PDUs mit einer bestimmten Eigenschaft gesendet werden, irgendwann eine dieser PDUs ausliefern.

Wie in allen Spezifikationen, die sich aus AMCs zusammensetzen, verwenden wir als Basis die AMCs *MSDUId* bzw. *ASDUId*, durch die den übertragenen PDUs Sequenznummern zugeordnet werden.

Die Erkennung von Übertragungsfehlern modellieren wir durch die AMCs *MCorruptions*, *ACorruptions*, *MPhantoms* und *APhantoms*, die auf die folgende Weise instantiiert werden:

```
usd ≙ XTPpdu;
tc ≙ { (p,q) | ( cmdNOCHECK(q.h.cmd) = 0 ∧
                 ( p = q ∨ q.h.check ≠ xsum(q.h.dlen + 32, q) ) ) ∨
               ( cmdNOCHECK(q.h.cmd) = 1 ∧
                 ( p.h = q.h ∨ q.h.check ≠ xsum(32, q.h) ) ) };
pd ≙ { p | ( cmdNOCHECK(p.h.cmd) = 0 ∧
             p.h.check = xsum(p.h.dlen + 32, p) ) ∨
           ( cmdNOCHECK(p.h.cmd) = 1 ∧
             p.h.check = xsum(32, p.h) ) };
```

Da die XTP-PDUs die Nutzdaten der Basiskanäle sind, ersetzen wir `usd` durch `XTPpdu`. Die Relation `tc` in den AMCs *MCorruptions* und *ACorruptions* beschreibt die Datenverfälschungen, die vom Dienstnehmer toleriert werden können. XTP-Instanzen können Verfälschungen relevanter Teile einer PDU immer tolerieren, wenn die Prüffunktion `xsum` die in Feld `check` enthaltene Prüfsumme nicht akzeptiert. Je nach Datentransfermodus wird dabei entweder die ganze PDU oder nur ihr PDU-Kopf überprüft. Im AMC *Phantoms* gibt der Parameter `pd` an, welche PDUs nicht Phantome sein dürfen. Der Parameter wird durch die Menge aller PDUs mit korrekter Prüfsumme ersetzt.

Das endliche „Gedchtnis“ der Basiskanäle wird durch die AMCs *MCapacity*, *ACapacity*, *MReorderings* und *AReorderingsMRq* sichergestellt. Danach darf eine Dateneinheit vom Basiskanal nur ausgeliefert werden, wenn nach ihrer Sendung weniger als `c + tr` weitere Dateneinheiten in Kanal M bzw. `tr` weitere Dateneinheiten in Kanal A gesendet wurden.

Um tragfähige Aussagen auch über die Eigenschaften der XTP-Datentransfermodi mit eingeschränkter Fehlererkennung bzw. Fehlerbehebung machen zu können, verwenden wir die AMCs *MRelativeCorruptions*, *ARelativeCorruptions*, *MRelativeGaps*, *ARelativeGaps*, *MRelativePhantoms* und *ARelativePhantoms*. Dadurch wird sichergestellt, daß der Anteil verfälschter oder verlorener Dateneinheiten sowie von Phantomen niemals bestimmte konstante Werte übersteigt, die durch die Parameter `maxrc`, `maxrg` und `maxrp` angegeben werden. In diesen FAPMs werden folgende Parameterersetzungen vorgenommen:

```
usdsize ≙
  [ p ∈ XTPpdu ↦ CASE cmdPFORMAT(p.h.cmd) = DATA → p.h.dlen,
                      cmdPFORMAT(p.h.cmd) = FIRST →
                        p.h.dlen - p.p.is.alen - p.p.is.tlen,
                      OTHERWISE → 0 ];
tcr ≙ { (p, q) | p = q };
```

```
pdr ≙ XTPpdu;
```

`usdsize` gibt die Anzahl an Nutzdatenbytes in der PDU an. Die Belegung von `tcr` und `pdr` legt fest, daß die relativen Obergrenzen von allen verfälschten PDUs bzw. Phantomen berücksichtigt werden.

Durch das AMC *LiveIn* wird die Aktivität der Basiskanäle garantiert. Es spezifiziert, daß eine PDU einer bestimmten Klasse, die durch die Relation `lrel` definiert ist, irgendwann ausgeliefert wird, wenn immer wieder PDUs dieser Klasse gesendet werden.

```
tg ≙ "infty";
lrel ≙ { (p,q) | p.p.is.ds = q.p.is.ds ∧
                 ( ( cmdNOCHECK(p.h.cmd) = 0 ∧
                     p.h.check = xsum(p.h.dlen + 32, p) ) ∨
                   ( cmdNOCHECK(p.h.cmd) = 1 ∧
                     p.h.check = xsum(32, p.h) ) ) };
```

Die Parameterbelegung von `lrel` legt fest, daß eine PDU ausgeliefert werden muß, wenn beliebig oft PDUs mit den gleichen Nutzdaten gesendet werden und die PDU eine korrekte Prüfsumme besitzt. Die anderen Parameter werden auf die gleiche Art wie in den Safety-AMCs instantiiert.

6.2.11 Instantiierung und Komposition der APMs

Die APMs unterscheiden sich von den FAPMs nur durch die Verwendung eindeutiger Sequenznummern aus einem unendlich großen Wertebereich. Wir erzeugen sie unter Verwendung derselben Parameterersetzungen, die wir bei den FAPMs vorgenommen haben. Lediglich für die FAPMs *AssignCNo*, *SelectCNo*, *FreeCNo* und *LiveFreeCNo* der Interconnection-Coordination werden keine APMs erzeugt.

6.3 Spezifikation des Transportdienstes

Die Spezifikation des Transportdienstes wird durch Instantiierung von SCs des Frameworks gebildet, die anschließend miteinander komponiert werden. Wir zeigen zunächst in den Abschnitten 6.3.1 und 6.3.2 die Bildung und Komposition der Datentransferkanäle. Die Erzeugung der Signalisierungskanalspezifikationen wird in den Abschnitten 6.3.3 und 6.3.4 erörtert. Danach diskutieren wir die Entwicklung der Intraconnection-Coordination-Dienstspezifikation in den Abschnitten 6.3.5 und 6.3.6. Schließlich wird die Instantiierung und Komposition der SCs der Interconnection-Coordination in den Abschnitten 6.3.7 und 6.3.8 erläutert.

6.3.1 Instantiierung der SCs des Datentransfers

In Abschnitt 6.2.2 stellten wir fest, daß XTP 18 funktionale Datentransfermodi anbietet, die durch die in Tabelle 6.1 beschriebenen Steuerflags `NOERR`, `NOCHECK`, `NOFLOW`, `RES` und `FASTNAK` gekennzeichnet werden. Die Flags `NOFLOW` und `RES` geben an, welche Flußkontrollmechanismen verwendet werden. Durch `FASTNAK` wird festgelegt, nach welcher Methode vom Empfänger erkannte Übertragungsfehler dem Sender von Daten mitgeteilt werden. Da jedoch sowohl die Flußkontrolle als auch die Meldung von Fehlern für den Transportdienstnehmer

transparent sind, erbringen alle Datentransfermodi, die sich nur in diesen Flags unterscheiden, denselben funktionalen Dienst. Somit kennt der von XTP erbrachte Transportdienst nur vier Datentransfermodi, die sich durch die Belegung der Steuerflags `NOERR` und `NOCHECK` unterscheiden. Wir erstellen Spezifikationen für diese vier Datentransferkanäle. Analog zum Protokoll werden die Kanäle durch die Flags `NOERR` und `NOCHECK` gekennzeichnet. In den Datentransferkanälen mit `NOERR = 0` sind Lücken im Datenstrom ausgeschlossen. Da der Datentransferdienst generell keine Verfälschungen, Duplikate und erkannte Phantome zuläßt, hat der Ausgabedatenstrom die gleiche Ordnung wie der Eingabedatenstrom, auch wenn die ausgelieferten Daten verfälscht sein können. Wenn `NOERR = 1` gesetzt ist, besteht hingegen die Möglichkeit von Lücken im Datenstrom, die jedoch eine bestimmte relative Häufigkeit nicht übersteigen dürfen. Durch das Flag `NOCHECK` wird angegeben, ob der Dienst die Auslieferung verfälschter Daten gestattet (`NOCHECK = 1`) oder nicht (`NOCHECK = 0`)[12]. Die Anzahl tolerierbarer Bitfehler in Datentransferkanälen mit `NOCHECK = 1` darf eine bestimmte relative Häufigkeit ebenfalls nicht überschreiten.

Zur Beschreibung der vier Transferkanäle instantiieren wir die SCs *SDUId*, *Corruptions*, *RelativeCorruptions*, *Duplicates*, *Reorderings*, *Gaps*, *RelativeGaps*, *Phantoms*, *Capacity* und *LiveInNoAttr* mit den folgenden Parameterersetzungen:

```
usd ≙ Byte;
tc ≙ { (p,p) | p ∈ Byte };
tcr ≙ { (p,p) | p ∈ Byte };
usdsize ≙ 1;
td ≙ 0;
tr ≙ 0;
tg ≙ 0;
pd ≙ SUBSET(Byte);
```

Da auch der Transportdienst byteweisen Datentransfer realisiert, erhält `usd`, das die Menge aller Nutzdaten angibt, den Wert `Byte`. Das SC *Corruptions* soll Verfälschungen von Daten generell ausschließen, was durch die Belegung von `tc` modelliert wird. Das SC *RelativeCorruptions*, das eine Obergrenze für die relative Häufigkeit von Verfälschungen garantiert, soll bei der Berechnung der relativen Häufigkeit alle übermittelten Dateneinheiten berücksichtigen und gleich gewichten. Deshalb erhalten die Parameter `tcr` und `usdsize` die angegebenen Belegungen. Wie schon gesagt wurde, sollen im Ausgabedatenstrom keine Duplikate und Vertauschungen auftreten. Aus diesem Grund haben die Parameter `td` und `tr` der SCs *Duplicates* und *Reorderings* den Wert 0. In allen Modi, in denen keine Lücken auftreten, wird das SC *Gaps* verwendet, dessen Parameter `tg` deshalb auch den Wert 0 annimmt. Die Belegung des Parameters `pd` im SC *Phantoms* stellt sicher, daß keine erkannten Phantome ausgeliefert werden.

6.3.2 Komposition der SCs des Datentransfers

Die Spezifikationen für die vier Datentransferkanäle werden aus den SCs auf die in der Tabelle 6.5 dargestellten Weise miteinander komponiert. Neben dem Basis-SC *SDUId* verwenden

[12] Reale Kommunikationsdienste können Datenverfälschungen niemals ganz ausschließen. `NOCHECK = 0` zeigt an, daß nur Verfälschungen auftreten, die durch die Bitfehlererkennung des den Dienst erbringenden Protokolls nicht erkannt werden können.

SC	0 0	0 1	1 0	1 1	NOCHECK NOERR
SDUId	⋆	⋆	⋆	⋆	
Corruptions	⋆	⋆			
RelativeCorruptions			⋆	⋆	
Duplicates	⋆	⋆	⋆	⋆	
Reorderings	⋆	⋆	⋆	⋆	
Gaps	⋆		⋆		
RelativeGaps		⋆		⋆	
Phantoms	⋆	⋆	⋆	⋆	
Capacity	⋆	⋆	⋆	⋆	
LiveInNoAttr	⋆	⋆	⋆	⋆	

Tabelle 6.5: Komposition der Datentransfer-SCs

alle Datenkanäle die SCs *Duplicates*, *Reorderings* und *Phantoms*, da Duplikate, Vertauschungen und Phantome im Ausgabedatenstrom nicht zulässig sind. *Capacity* stellt sicher, daß der Dienst nur eine bestimmte Anzahl gesendeter aber nicht empfangener Dateneinheiten enthält. Die Liveness der Datentransferkanäle wird durch das SC *LiveInNoAttr* spezifiziert. Erwartungsgemäß tritt das SC *Corruptions*, das den Ausschluß von Datenverfälschungen beschreibt, nur in den beiden Kanälen mit `NOCHECK = 0` auf. In den anderen Kanälen wird durch das SC *RelativeCorruptions* nur eine Obergrenze der relativen Häufigkeit an Verfälschungen festgelegt. In den Kanälen mit `NOERR = 0` werden durch `Gaps` Lücken im Datenstrom ausgeschlossen. Dagegen wird in den Modi mit `NOERR = 1` durch `RelativeGaps` eine maximal tolerierbare relative Häufigkeit an Lücken im Datenstrom gefordert.

6.3.3 Instantiierung der SCs der Signalisierungskanäle

Wie in der XTP-Protokollbeschreibung (siehe Abschnitt 6.2.4) verwenden wir auch in der Dienstbeschreibung drei Kanaltypen `"First"`, `"Active"` und `"Last"` zur Spezifikation des Transfers von Signalisierungsdaten, die von einem Dienstnehmer initiiert werden. Diese Dienste werden in `"First"` und `"Last"` mit Hilfe der SCs *SDUId*, *Corruptions*, *Duplicates*, *Reorderings* und *Phantoms* modelliert. Der Kanaltyp `"Active"` verwendet *Duplicates*, *Reorderings* und *Phantoms* ebenfalls. Außerdem enthält dieser Kanaltyp vier weitere SCs, die mit Hilfe des neu eingeführten FAPMs *SSynchronize* (siehe Abschnitt 6.2.4) realisiert sind. Diese SCs werden weiter unten in diesem Abschnitt vorgestellt. Die Parameter der SCs werden auf die folgende Weise ersetzt:

```
SigdTypes = { "FirstDuplex", "FirstSimplex", "NewQ", "WClose",
              "Final", "UAbort" };
sigd ≙ [ sTYPE : SigdTypes, sNOERR : {0, 1}, sNOCHECK : {0, 1},
         sNOFLOW : {0, 1}, sRES : {0, 1}, sFASTNAK : {0, 1},
         sSEQ : DWord, sDESTADR : { x | x ∈ Nat ∧ x < 2^(2^19 − 2^5 − 1)} ];
usd ≙ sigd;
tc ≙ { (p,p) | p ∈ sigd };
```

```
td ≙ 0;
tr ≙ 0;
pd ≙ SUBSET(sigd);
```

Somit sind in den drei Kanälen Datenverfälschungen, Datenvertauschungen, Duplikate und Phantome ausgeschlossen.

Ein vierter Kanaltyp modelliert den Transport von Information über einen vom Diensterbringer initiierten Verbindungsabbruch (Provider-Abort). Er erfüllt die Safety-SCs *SDUId* und *Phantoms* sowie das Liveness-SC *LiveInNoAttr*. Dadurch ist sichergestellt, daß der Dienst beim Partner tatsächlich angezeigt wird. Da dieser Kanal nur für eine einzige Übertragung eingesetzt wird, für die außer der Dienstanzeige keine zusätzliche Steuerinformation notwendig ist, kann auf die Einhaltung des SCs *Corruptions* verzichtet werden. In SC *Phantoms* wird der Parameter `pd` durch den Wert `Any` ersetzt, um die Auslieferung von Phantomen zu verhindern.

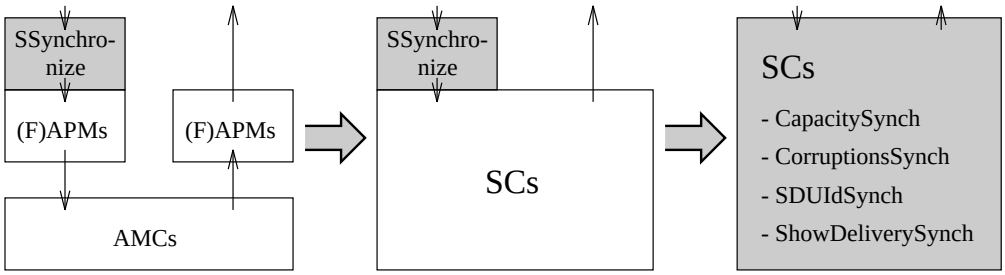

Abbildung 6.4: Verifikation der SCs für FAPM *SSynchronize*

Im folgenden stellen wir die vier neuen SCs *SDUIdSynch*, *CorruptionsSynch*, *CapacitySynch* und *ShowDeliverySynch* vor, die durch das in Abschnitt 6.2.4 eingeführte FAPM *SSynchronize* und andere APMs sowie AMCs erbracht werden. *SSynchronize* wird verwendet, um die XTP-spezifische Sequenznummervergabe bei Verhandlungen der Datenübertragungsrate modellieren zu können. Damit *SSynchronize* zusammen mit den anderen FAPMs des Datentransfers verwendet werden kann, wurde es als Filter entwickelt, der an der XTP-Dienstschnittstelle auf der Seite des Senders angeordnet wird (siehe linke Seite in Abbildung 6.4).

Diese Anordnung vereinfacht den Beweis der für die vier SCs neu zu bildenen Theoreme, da man ihn in zwei Schritte aufteilen kann (siehe Abbildung 6.4). Neben den aus *SSynchronize* und weiteren FAPMs bestehenden Feinspezifikationen und den SCs werden Spezifikationen eines mittleren Abstraktionsgrads gebildet, die aus SCs des Frameworks und einem *SSynchronize* entsprechenden APM *SSynchronize* zusammengesetzt sind. In einem ersten Schritt wird unter Verwendung von Theoremen des Frameworks bewiesen, daß eine dieser mittleren Spezifikationen durch eine Feinspezifikation erfüllt ist. Im zweiten Schritt wird gezeigt, daß die mittlere Spezifikation ein SC implementiert. Dieser zweite Schritt ist einfach, da keine Verteilungsaspekte berücksichtigt werden müssen.

Im nachfolgenden stellen wir die neuen SCs kurz vor und skizzieren, wie man sie verifiziert, indem wir das verwendete Refinement Mapping und eine für den Beweis notwendige Invariante angeben. Das SC *SDUIdSynch* modelliert analog zum SC *SDUId* den Sachverhalt,

daß unerkannte Phantome nicht auftreten dürfen. Es ist sichergestellt, daß alle ausgelieferten Daten Sequenznummern besitzen, die um höchstens `tmax` größer als die Sequenzzahl des zuletzt beim Sender bestätigten Dienstes sind oder ein erkanntes Phantom anzeigen.

```
PROCESS SDUIdSynch    Sequenznummernvergabe, Alle Phantome sind erkennbar
                  ( tmax : Nat )     Maximale Anzahl an Wiederholungen
IMPORT Constants;
BODY
  VARIABLES
    min : Nat ∪ {"notsent"}; Sequenznummer des ersten Signals
  INIT ≜ min = 0 ;
  ACTIONS
    SIn (krq : Nat ∪ {"notsent"}) ≜ Auslieferung von Signalisierungsinformation an den Dienstn.
      ( (krq = "notsent") ∨ krq < min + tmax ) ∧
      min′ = min;
    SCf (krq : Nat ∪ {"notsent"}) ≜
      Implizite Bestaetigung der Auslieferung von Signalisierungsinformation an den Dienstnehmer
      min′ = krq + 1 ;
END
```

Für den Beweis betrachten wir die Kombination des APM *SSynchronize* und des SC *SDUId*. Der Beweis wird als TLA-Safetybeweis unter Verwendung des folgenden Refinement Mappings `RM` und der Invariante `Inv` geführt:

```
RM ≜ SDUIdSynch.min = SSynchronize.min;
Inv ≜ ∀ k ∈ SSynchronize.obuf ::
        k < SSynchronize.smin + tmax ∧
      SSynchronize.cRq = SDUId.cRq ∧
      SDUId.cRq ≤ SSynchronize.smin + tmax;
```

Das SC *CorruptionsSynch* modelliert die Korrektheit der übertragenen Daten.

```
PROCESS CorruptionsSynch    Sicherung gegen Datenverfaelschungen
                          ( tmax : Nat )     Maximale Anzahl an Wiederholungen
IMPORT Constants;
BODY
  VARIABLES
    obuf : SUBSET(Nat ∪ {"notsent"} × sig);
  INIT ≜ obuf = ∅;    Menge gesendeter Signale
  ACTIONS
    SRq (krq : Nat ∪ {"notsent"}; s : sig) ≜
      Uebergabe von Signalisierungsinformation durch den Dienstnehmer
      obuf′ = obuf ∪ { (k,s) | krq ≤ k < krq + tmax} ;
    SIn (krq : Nat ∪ {"notsent"}; s : sig) ≜
      Auslieferung von Signalisierungsinformation an den Dienstnehmer
      ( krq = "notsent" ∨ ∀ t ∈ sig :: ((krq,t) ∉ obuf) ∨
        (krq,s) ∈ obuf ) ∧
      obuf′ = obuf ;
    SCf (krq : Nat ∪ {"notsent"}) ≜
      Implizite Bestaetigung der Auslieferung von Signalisierungsinformation an den Dienstnehmer
      obuf′ = obuf \ { (k,s) | k > krq };
END
```

Der Beweis wird mit Hilfe der Kombination aus *SSynchronize* und den SCs *SDUId* sowie *Corruptions* erbracht:

```
RM ≙ CorruptionsSynch.obuf = SSynchronize.sobuf;
Inv ≙ SSynchronize.cRq = SDUId.cRq ∧
      ∀ k ∈ Corruptions.buf :: k < SSynchronize.cRq ∧
      Corruptions.buf ⊆ SSynchronize.sobuf;
```

Durch das SC *CapacitySynch* wird sichergestellt, daß der Dienstnutzer eine erneute Dienstanforderung erst nach Bestätigung des aktuellen Dienstes anfordert.

```
PROCESS CapacitySynch      Kapazitaet des Dienstes
IMPORT Constants;
BODY
 VARIABLES
    sst : {"idle","active"}; Signal in Uebertragung?
  INIT ≙ sst = "idle" ;
  ACTIONS
    SRq ≙ Uebergabe von Signalisierungsinformation durch den Dienstnehmer
      sst = "idle" ∧ sst' = "active";
    SCf ≙ Implizite Bestaetigung der Auslieferung von Signalisierungsinformation an den Dienstn.
      sst' = "idle";
END
```

Für den Beweis reicht das APM *SSynchronize* und das folgende Refinement Mapping aus:

```
RM ≙ CapacitySynch.sst = IF (∃ t :: (SSynchronize.smin,t) ∈ SSynchronize.obuf)
                         THEN "active" ELSE "idle";
```

Schließlich definieren wir das SC *ShowDeliverySynch*, durch das sichergestellt wird, daß nur an den Dienstnehmer ausgelieferte Signalisierungsinformation bestätigt wird:

```
PROCESS ShowDeliverySynch  Bestaetigung nur von ausgelieferter Signalisierungsinformation
IMPORT Constants;
BODY
  VARIABLES
    smaxIn : Nat ∪ {"notsent"};    Hoechste Sequenznummer der ausgelieferten Daten
  INIT ≙ smaxIn = 0;
  ACTIONS
    SIn ( krq : Nat ∪ {"notsent"} ) ≙ Auslieferung von Signalisierungsinformation an den Dienstn.
      smaxIn' = IF (krq = "notsent")
                THEN smaxIn ELSE Max(smaxIn, krq + 1);
    SCf ( krq : Nat ∪ {"notsent"} ) ≙
      Implizite Bestaetigung der Auslieferung von Signalisierungsinfo. an den Dienstnehmer
      krq = smaxIn - 1 ∧
      smaxIn' = smaxIn;
  END
```

Zum Beweis verwenden wir die Kombination von *SSynchronize* mit den SCs *SDUId* und *ShowDelivery* mit dem Refinement Mapping und der Invarianten

```
RM ≙ ShowDeliverySynch.smaxIn = ShowDelivery.smaxIn;
Inv ≙ SSynchronize.cRq = SDUId.cRq ∧
      ShowDelivery.smaxin ≤ SDUId.cRq;
```

SC	"First"	"Active"	"Last"
SDUId	⋆		⋆
Corruptions	⋆		⋆
Reorderings	⋆	⋆	⋆
Phantoms	⋆	⋆	⋆
SDUIdSynch		⋆	
CorruptionsSynch		⋆	
CapacitySynch		⋆	
ShowDeliverySynch		⋆	

SC	Prov.-Ab.
SDUId	⋆
Phantoms	⋆
LiveInNoAttr	⋆

Tabelle 6.6: Komposition der SCs der Signalisierungsdatenkanäle

6.3.4 Komposition der SCs der Signalisierungskanäle

Die Komposition der vier Datenkanaltypen ist in der Tabelle 6.6 beschrieben. Die Spezifikationen der Datenkanäle `"First"` und `"Last"` werden durch Komposition der SCs *SDUId*, *Corruptions*, *Reorderings* und *Phantoms* gebildet. Dagegen werden *SDUId* und *Corruptions* in Kanaltyp `"Active"` durch die SCs *SDUIdSynch* und *CorruptionsSynch* ersetzt. Außerdem enthält diese Spezifikation die SCs *CapacitySynch* und *ShowDeliverySynch*. Die Spezifikation für den Provider-Abort-Kanaltyp setzt sich aus den SCs *SDUId*, *Phantoms* und *LiveInNoAttr* zusammen.

6.3.5 Instantiierung der SCs der Intraconnection-Coordination

Die SCs *ConnectionState*, *DataChanOpenW*, *DataChanOpenR*, *StopOld* und *UserBehaviour* der Intraconnection-Coordination beschreiben nur Aspekte der Steuerung einzelner Dienstzugangspunkte und verwenden dieselben Parameterersetzungen wie die in Abschnitt 6.2.6 vorgestellten gleichnamigen FAPMs. Außerdem werden die SCs *DelayedClose* und *LivePAbI* verwendet, wobei der Parameter `tg` in *DelayedClose* durch den Wert 0 ersetzt wird.

6.3.6 Komposition der SCs der Intraconnection-Coordination

Wie bei der Bildung der Protokollspezifikation (siehe Abschnitt 6.2.7) wird für jeden der beiden Dienstzugangspunkte der Verbindung jeweils eine Spezifikation aus Intraconnection-Coordination-Komponenten gebildet. An jedem Dienstzugangspunkt werden die vier Datenkanäle mit je einer Kopie der SCs *DataChanOpenW*, *DataChanOpenR*, *DelayedClose* und *StopOld* gekoppelt. Zusätzlich erhält jeder Dienstzugangspunkt je eine Kopie der SCs *ConnectionState*, *UserBehaviour* und *LivePAbI*.

Die Spezifikation einer Transportverbindung setzt sich aus jeweils einer Instanz aller acht Daten- und Signalisierungskanäle für jede Senderichtung und den oben genannten Spezifikationen der Intraconnection-Coordination zusammen. Die Menge aller möglichen Verbindungen im Transportsystem wird durch ein cTLA$^+$-Prozeßfeld über die Verbindungsspezifikation beschrieben.

6.3.7 Instantiierung der SCs der Interconnection-Coordination

Der Dienst besteht aus den SCs *OpenConnections* und *FindDestination*, in denen die gleichen Parameterersetzungen wie in Abschnitt 6.2.8 vorgenommen werden.

6.3.8 Komposition der SCs der Interconnection-Coordination

Die Komposition der beiden SCs *OpenConnections* und *FindDestination* bildet die Spezifikation der Verwaltung aller Dienstzugangspunkte eines Rechnernetzknotens. Aus dieser Spezifikation wird ein cTLA$^+$-Prozeßfeld aller Rechnernetzknoten gebildet, das mit dem Prozeßfeld der Verbindungsspezifikationen zu der Spezifikation der Transportdienstschnittstelle des gesamten Transportsystems komponiert wird.

6.4 Verifikation der Dienstspezifikation

Zunächst zeigen wir in Abschnitt 6.4.1, daß die aus FAPMs und AMCs bestehende Protokollspezifikation von XTP die abstraktere Protokollspezifikation erfüllt, die sich aus APMs und AMCs zusammensetzt. Wie in Abschnitt 5.5.1 erläutert wurde, müssen wir dazu zeigen, daß man für alle aktiv in Übertragung befindlichen Dateneinheiten eindeutige Sequenznummern vergeben kann. Außerdem müssen wir sicherstellen, daß alle aktiven Verbindungen in XTP durch eindeutige Kennungen gekennzeichnet werden können. Anschließend zeigen wir in Abschnitt 6.4.2 unter Verwendung der Theoreme des Frameworks, daß die SCs der Dienstspezifikation durch die APMs und AMCs der abstrakten Protokollspezifikation implementiert werden.

6.4.1 Verifikation der APMs

Die APMs unterscheiden sich von den FAPMs durch die Verwendung eindeutiger Sequenznummern aus einem unendlich großen Wertebereich. In Abschnitt 5.5.1 zeigten wir, daß man die APMs durch FAPMs implementieren kann, wenn die untersuchte Protokollspezifikation die FAPMs *SBufferKey*, *RBufferKey*, *SAcknowledge*, *SCapacity*, *RAcknowledge* und *SendInOrder* enthält. Diese FAPMs sind in der XTP-Spezifikation in den Beschreibungen aller Datentransferkanaltypen und des Signalisierungskanaltyps `"Active"` enthalten, der als einziger Kanaltyp der Verbindungsverwaltung beliebig viele Sequenznummern verwendet (siehe Tabellen 6.2 und 6.4). Zusätzlich muß sichergestellt sein, daß der Basisdienst nicht beliebig alte PDUs ausliefert. Wir müssen nun nachweisen, daß die Anzahl `ir` der Sequenznummern in XTP mindestens so groß wie der Wert

$$2 \cdot \mathtt{sc} + 2 \cdot \mathtt{rcc} + 2 \cdot \mathtt{rcu} + \mathtt{mb} \cdot (\mathtt{c} + \mathtt{tr})$$

ist. `sc` bezeichnet die Größe des Sendepuffers und `rcc + rcu` die Größe des Empfangspuffers. `c + tr` legen die maximale Anzahl der zu einem Zeitpunkt auslieferbaren PDUs fest. `mb` bestimmt, wieviele Dateneinheiten in einer PDU maximal gesendet bzw. bestätigt werden dürfen. `ir`, das für die Größe des Wertebereichs der Sequenzzahlen steht, hat in den Datentransferkanaltypen von XTP den Wert 2^{64} und in den Signalisierungskanaltypen den Wert 2^{32}.

Die Größe der Puffer und die maximale Verweildauer von PDUs im Basisdienst hängt in der Realität von der XTP-Implementierung und den Eigenschaften des eingesetzten Basisnetzwerks ab, so daß man den Nachweis, daß genügend eindeutige Sequenznummern zur Verfügung stehen, eigentlich für jedes reale Transportsystem separat führen müßte. Der Wertebereich der Sequenzzahlen ist in XTP jedoch so groß gewählt, daß sie für alle heute und in näherer Zukunft verfügbaren Rechnernetze und Stationen ausreicht. Das gilt insbesondere für die Datentransferkanäle, für die 2^{64} Sequenznummern zur Verfügung stehen. Wenn man die Hälfte von 2^{64} Bytes für die Datenpuffer reserviert und gleich große Sende- und Empfangspuffer annimmt, kann man für jeden Puffer 2^{62} Bytes (vier Exabytes oder vier Milliarden Gigabytes) reservieren. Die übrigen 2^{63} Bytes werden zur Sicherung gegen die Auslieferung veralteter PDUs durch die Datentransferkanäle verwendet. Wir vernachlässigen die Protokollsteuerinformation und gehen von einem kontinuierlichen Strom gesendeter Daten bei einer Datenübertragungsrate von 8 GBit/s aus. Dann dauert die Sendung dieser 2^{63} Bytes 256 Jahre[13]. Das Basisnetz muß also garantieren, daß alle ausgelieferten Nutzdaten einer Verbindung vor weniger als 256 Jahren gesendet wurden. Wir nehmen diese Forderung als gegeben an.

Eine höhere Wahrscheinlichkeit, Sequenznummern wiederverwenden zu müssen, besteht hingegen beim Signalisierungskanal `"Active"`, da hier der Sequenznummernbereich in PDU-Feld `sync` nur aus vier Bytes besteht. Die Sequenznummern werden für die Aushandlung der Datenübertragungsrate und des Datentransfermodus verwendet. Die Puffergröße hierfür ist begrenzt, da der Sendepuffer nur die relativ kleine Anzahl von `tmax` PDUs (siehe FAPM *SSynchronize*) und der Empfangspuffer nur eine einzige PDU speichern muß. Wir vernachlässigen die Größe der Puffer und nehmen vereinfachend an, daß die Signalisierung korrekt funktioniert, wenn der Signalisierungskanal `"Active"` garantiert, daß nach Auslieferung einer PDU nur noch höchstens die letzten 2^{32} zuvor gesendeten PDUs ausgeliefert werden können. Bei der Aushandlung der Übertragungsrate werden immer PDUs des XTP-Formats `TCNTL` eingesetzt, die eine Länge von mindestens 72 Bytes besitzen. Die Übertragung von 2^{32} PDUs benötigt dann aber bei einer Datenübertragungsrate von 8 GBit/s mindestens 288 Sekunden. Das ist in Hochleistungsrechnernetzen zwar eine sehr lange Zeitspanne, jedoch gerade bei Fernverkehrsübertragungen mit vielen zwischenspeichernden Übertragungsknoten nicht völlig unrealistisch. Allerdings berücksichtigt diese Argumentation nicht, daß in XTP nur `tmax`-viele PDUs nacheinander ohne eine Bestätigung gesendet werden dürfen. Zum Beispiel erfolgen 2^{28} Handshakes zwischen den Verbindungspartnern, bis die Sequenznummer 0 wiederverwendet werden muß, wenn `tmax` den Wert 16 hat. Nimmt man auf einer Verbindung von 4885 km eine Turn-Around-Zeit von 2^{-4} Sekunden an, benötigt die Ausführung dieser Handshakes etwa ein halbes Jahr. Bei kürzeren Entfernungen sinkt zwar die Turn-Around-Zeit, jedoch wird auch die Wahrscheinlichkeit der Auslieferung veralteter PDUs geringer. Wir gehen somit auch bei Signalisierungskanal `"Active"` davon aus, daß in allen im Moment und in der näheren Zukunft existierenden Hochleistungsrechnernetzen die Lebenszeit alter PDUs soweit beschränkt ist, daß keine Protokollfehler aufgrund verspäteter Auslieferungen von PDUs auftreten können.

Wir nehmen somit

$$2 \cdot \mathtt{sc} + 2 \cdot \mathtt{rcc} + 2 \cdot \mathtt{rcu} + \mathtt{mb} \cdot (\mathtt{c} + \mathtt{tr}) \leq \mathtt{ir}$$

[13] 1 Jahr wird durch 2^{25} Sekunden abgeschätzt.

an und können unter Verwendung der in Abschnitt 5.7.1 eingeführten Theoreme zeigen, daß die instantiierten APMs des Datentransfers und der Intraconnection-Coordination von den FAPMs erfüllt werden.

Auch die APMs der Interconnection-Coordination werden durch die FAPMs implementiert. Da die XTP-Spezifikation der Interconnection-Coordination (siehe Abschnitt 6.2.8) die FAPMs *AssignCNo*, *SelectCNo*, *FreeCNo* und *LiveFreeCNo* enthält, kann der Verfeinerungsbeweis mit Hilfe der in Abschnitt 5.7.2 aufgeführten Theoreme erbracht werden. Der Beweis, daß die Kopplung der APMs derjenigen der FAPMs entspricht, ist trivial, da in beiden Spezifikationen nur Aktionen mit denselben Bezeichnern miteinander gekoppelt werden. Nach der Definition des Frameworks sind die Kopplungen der FAPMs und der APMs dann aber zueinander konsistent (siehe Kapitel 5).

6.4.2 Verifikation der SCs

Der Beweis der SCs erfolgt unter Verwendung der in Abschnitt 5.6 vorgestellten Theoreme. Wir suchen für jede Instanz eines SCs ein geeignetes Theorem, dessen Subsystem `Sys` ein Teil der aus APMs bestehenden abstrakten XTP-Protokollspezifikation ist. Wir ersetzen die Prozeßtypparameter in den Theoremen durch dieselben Werte wie die Prozeßtypen (siehe Abschnitte 6.2.11 und 6.3). Dann überprüfen wir, ob die Parameterbedingung `Pars` im Theorem eingehalten wird. Eine explizite Überprüfung der Umgebungsbedingung `EnvCond` in den Livenesstheoremen ist nicht notwendig, da für alle Frameworkprozesse außer dem Intraconnection-Coordination-APM *DataChanOpenR* sichergestellt wurde, daß sie `EnvCond` nicht verletzen. Auch das APM *DataChanOpenR* verletzt diese Bedingung nicht, da jeder Kanal in der Protokollspezifikation, der *DataChanOpenR* enthält, auf einen Kanal in der Dienstspezifikation abgebildet wird, in dem das SC *DataChanOpenR* ebenfalls gekoppelt ist. Das einzige APM, das nicht im Framework enthalten ist, ist *SSynchronize*. Für dieses APM haben wir jedoch ebenfalls nachgewiesen, daß es `EnvCond` in den Livenesstheoremen erfüllt.

Als Beispiel geben wir den Beweis des Datentransfer-SCs *Corruptions* mit der Parameterinstantiierung $\texttt{tc} \triangleq \{(p,p) | p \in \texttt{Byte}\}$ an. Dazu wird ein Theorem zum Beweis des SCs *Corruptions* verwendet, das nach der Ersetzung der Parameter das folgenden Aussehen besitzt[14]:

```
LET Pars ≜ Anforderungen an die Prozesstypparameter
           { (p,q) | ( cmdNOCHECK(q.h.cmd) = 0 ∧
                       ( p = q ∨ q.h.check ≠ xsum(q.h.dlen + 32, q) ) ) ∨
                     ( cmdNOCHECK(q.h.cmd) = 1 ∧
                       ( p.h = q.h ∨ q.h.check ≠ xsum(32, q.h) ) ) } ⊆
             { (p,q) | ( ( cmdNOCHECK(q.h.cmd) = 0 ⇒
                                q.h.check ≠ xsum(q.h.dlen + 32, q) ) ∧
                         ( cmdNOCHECK(q.h.cmd) = 1 ⇒
                             q.h.check ≠ xsum(32, q.h) ) ) ∨
                       ( ( ( cmdPFORMAT(p.h.cmd) = DATA ⇒
                             ( p.h.seq = q.h.seq ∧ p.h.dlen = q.h.dlen ) ∧
                             ( ∀ n ∈ DWord ::
                                 between(p.h.seq, n,
                                         (p.h.seq + p.h.dlen) mod DWord ) ⇒
                                   p.p.is.ds = q.p.is.ds ) ) ) ∧
                         ( cmdPFORMAT(p.h.cmd) = FIRST ⇒
```

[14] Zur Erlangung einer besseren Übersicht wurden in `Pars` einige Umformungen vorgenommen.

```
                                ( p.h.dlen - p.p.is.alen - p.p.is.tlen =
                                  q.h.dlen - q.p.is.alen - q.p.is.tlen ) ∧
                                ( ∀ n ∈ DWord ::
                                    n < p.h.dlen - p.p.is.alen - p.p.is.tlen ⇒
                                      p.p.is.ds = q.p.is.ds ) ) ) };
Sys ≙  Subsystem aus APMs und AMCs, durch das LiveIn erfuellt wird
       SBufferKey (mb, NOERR, NOCHECK, NOFLOW, RES, FASTNAK) ∧
       SBufferUsd ∧
       RBufferKey (rcc, rcu, NOERR, NOCHECK, NOFLOW, RES, FASTNAK) ∧
       RBufferUsd ∧
       MSDUId ∧
       MCorruptions ∧
       MPhantoms ∧
       CC_Corruptions;
IN Pars ∧ Sys ⇒ Corruptions({ (p, p) | p ∈ Byte })     Theorem
```

Die in `Sys` aufgeführten APMs und AMCs sind in den Spezifikationen aller abstrakten XTP-Datentransferkanäle enthalten, so daß `Sys` eine Teilspezifikation der abstrakten XTP-Protokollspezifikation ist. Für den Fall, daß `cmdNOCHECK(q.h.cmd)` $= 0$ ist, gilt zudem das Prädikat `Pars`. `Pars` beschreibt eine Teilmengenbeziehung der Art $M_1 \subseteq M_2$. Für alle Elemente `(p,q)` in M_1 gilt `q.h.check` $\neq$ `xsum(q.h.dlen + 32, q)` oder `p = q`. Die Elemente der ersten Gruppe sind nach der Definition auch Elemente von M_2. Alle Elemente aus M_1 mit `p = q` haben die Eigenschaft, daß alle Feldkomponenten von `p` und `q` gleich sind. Da das insbesondere auch für die Feldkomponenten `h.seq`, `h.dlen`, `p.is.ds`, `p.is.alen` und `p.is.tlen` gilt, sind diese Elemente nach Definition ebenfalls Elemente von M_2. Für `cmdNOCHECK(q.h.cmd)` $= 1$ gilt `Pars` hingegen nicht. Allerdings verwenden wir das Theorem auch nur zum Beweis der Datentransferkanäle, in deren PDUs das Flag `NOCHECK` nicht gesetzt ist.

Auf die gleiche Art wird für jedes SC der Dienstspezifikation unter Verwendung von Theoremen des Frameworks gezeigt, daß es durch ein Subsystem der Protokollspezifikation implementiert wird. Diese Beweise werden hier nicht im einzelnen dargestellt.

Zum Abschluß müssen wir noch sicherstellen, daß nur gleichnamige SCs bzw. APMs miteinander gekoppelt wurden, um die Konsistenz dieser Kopplungen garantieren zu können. Wie in den Abschnitten 6.2 und 6.3 gezeigt wurde, war das immer der Fall.

6.5 Bewertung der XTP-Qualität

In den vorhergehenden Abschnitten haben wir auf der Basis des Frameworks und unter Berücksichtigung des PDU-Formats von XTP SCs und FAPMs erstellt, die zu einer Dienst- und einer Protokollspezifikation komponiert wurden. Da die XTP-Beschreibung [XTP95] keine Dienstschnittstellenbeschreibung besitzt, mit der man die Dienstspezifikation vergleichen kann, haben wir nur die Möglichkeit zu untersuchen, ob der spezifizierte Dienst für den Einsatz in der Hochleistungskommunikation sinnvoll ist. Diese Untersuchung wird in Abschnitt 6.5.1 vorgenommen. Anschließend wird in Abschnitt 6.5.2 überprüft, ob die Safety-FAPMs zu den in der XTP-Beschreibung in natürlicher Sprache angegebenen Verhaltensbeschreibungen konform sind. Ein von uns aufgedeckter Fehler von XTP wird in Abschnitt 6.5.3 erläutert. Abschließend wird in Abschnitt 6.5.4 untersucht, ob XTP Verhalten toleriert, durch das die in den Liveness-FAPMs spezifizierten Fairnessanforderungen verletzt werden.

6.5.1 Eigenschaften der SCs

Wir vergleichen im folgenden die Eigenschaften des von XTP angebotenen Transportdienstes mit den in Abschnitt 2.2 aufgeführten Anforderungen an Hochleistungstransportdienste. Wie schon in Abschnitt 6.4.2 festgestellt wurde, hängen die durch die von uns erzeugten SCs modellierten Dienstleistungen im Gegensatz zu den Protokollmechanismen nur vom Vorhandensein von Fehlerbehebungsfunktionen (XTP-Flag `NOERR`) und Bitfehlererkennungsfunktionen (`NOCHECK`) ab (siehe Tabelle 6.5). Die Auslieferung von stets unverfälschten Daten (SC *Corruptions*) wird nur durch die Kanaltypen sichergestellt, die eine Bitfehlererkennung des gesamten Bereichs einer PDU besitzen (`NOCHECK = 0`). Ansonsten kann nur garantiert werden, daß die relative Anzahl an Verfälschungen nicht eine von den Basiskanälen garantierte Grenze überschreitet (SC *RelativeCorruptions*).

Der Ausgabestrom in XTP kann in einigen Datenkanälen verlustbehaftet sein. Lücken sind nur ausgeschlossen (SC *Gaps*), wenn der Kanaltyp die Behebung von Fehlern anbietet (`NOERR = 0`). In diesem Fall kann eine bei der Übertragung verlorengegangene oder verfälschte Dateneinheit erneut übertragen werden. Ansonsten kann man nur eine obere Grenze für die relative Anzahl von Lücken angeben (SC *RelativeGaps*), die sich aus der vom Übertragungsmedium garantierten maximalen Anzahl an Verlusten bzw. an Bitfehlern ergibt. Dagegen ist der Ausgabedatenstrom immer geordnet (SC *Reorderings*) und enthält keine Duplikate (SC *Duplicates*).

Die Auslieferung von Phantomen (SC *Phantoms*) ist ausgeschlossen, wenn die Bitfehlerbehebung die gesamte PDU betrifft (`NOCHECK = 0`). Sonst wird die Anzahl von Phantomen durch die von den Basiskanälen garantierte Rate begrenzt (SC *RelativePhantoms*).

Die Existenz einer endlichen Systemkapazität (SC *Capacity*) und die Garantie, daß bis auf durch die SCs *Gaps* oder *RelativeGaps* tolerierten Lücken alle gesendeten Dateneinheiten dem empfangenden Dienstnehmer ausgeliefert werden (SC *LiveIn*), sind weitere typische Transportdiensteigenschaften. In Bezug auf den Datentransfer erfüllt der spezifizierte Dienst also die Erwartungen.

Auch die Dienste, die durch die Signalisierungskanaltypen erbracht werden, entsprechen mit einer Ausnahme den Erwartungen. In allen drei Kanaltypen ist sichergestellt, daß die übertragenen Daten unverfälscht sind (SCs *Corruptions* und *CorruptionsSynch*) und nicht vertauscht werden (SC *Reorderings*). Außerdem können keine Phantome ausgeliefert werden (SC *Phantoms*). Allerdings können aufgrund der XTP-Fehlerbehebungsmechanismen die Dienste zur Aushandlung der Übertragungsrate und zur Schließung von Datentransferkanälen im Kanaltyp `"Active"` dupliziert ausgeliefert werden. Das SC *SDUIdSynch* stellt lediglich eine Obergrenze für die Anzahl an Duplikaten sicher. Das entspricht jedoch einem von den XTP-Entwicklern vorgesehenen Verhalten (vgl. [XTP95]), das nicht zu schwerwiegenden Protokollfehlern führt.

Alle Kanalspezifikationen sehen vor, daß zu einem Zeitpunkt immer nur ein Dienst aktiv ist, dessen Daten über den entsprechenden Kanal übertragen werden (SC *CapacitySynch* in Kanaltyp `"Active"`; die anderen Kanaltypen erlauben nur die Übertragung einer einzigen Signalisierungsdateneinheit).

Im Gegensatz zu den Datentransferkanälen wird nicht garantiert, daß gesendete Dateneinheiten ausgeliefert werden. Doch wird eine Verbindung abgebrochen, wenn ein Signal permanent nicht ausgeliefert werden kann (SC *LivePAbI*). Dieses Verhalten ist für die Verbindungsverwaltung eines Transportdienstes ebenfalls typisch.

Da Benachrichtigungen über Provider-Aborts keine weitere Signalisierungsinformation zulassen und innerhalb einer Verbindung auch höchstens von einer Station einmal initiiert werden dürfen, garantiert der entsprechende Kanaltyp auch nur den Ausschluß von Phantomen (SC *Phantoms*). Zusätzlich wird die Auslieferung des Dienstes beim Partner garantiert (SC *LiveIn*).

Die SCs der Intraconnection-Coordination und der Interconnection-Coordination sind auf die gleiche Art zusammengesetzt wie die bei der Protokollmodellierung verwendeten FAPMs[15], deren Konformität wir im folgenden Abschnitt untersuchen.

Als Fazit dieser Betrachtung können wir somit festhalten, daß die Funktionalität des durch die SCs spezifizierten Transportdienstes den Erwartungen an die Transportdienstschnittstelle eines Hochleistungsprotokolls entspricht.

6.5.2 Konformität der FAPMs

Wie in der Einleitung dieses Kapitels schon erwähnt wurde, erschwert der informelle Charakter der XTP-Beschreibung [XTP95] die Bewertung der FAPMs. Wir untersuchen deshalb jedes FAPM auf Widersprüche gegenüber der XTP-Ablaufbeschreibung.

Bei der Analyse der FAPMs des Datentransfers ist die Konformität der FAPMs *SBufferKey*, *SBufferUsd*, *RBufferKey* und *RBufferUsd* zu [XTP95] einfach nachzuweisen. Aus der Beschreibung der Paketformate `FIRST` und `DATA` (vgl. [XTP95], Kapitel 3) geht hervor, daß jede Daten-PDU eine zusammenhängende Folge an Bytes enthält, deren Sequenznummern gemäß der Spezifikationen verwaltet werden. Auch die in FAPM *RBufferKey* modellierte Bitfehlererkennung entspricht der Vorgabe (vgl. [XTP95], Abschnitt 4.6.1). Die Protokollmechanismen zur Bestätigung korrekt empfangener Daten (FAPMs *SAcknowledge*, *RAcknowledge*) erfüllen die Beschreibung (vgl. [XTP95], Abschnitt 4.6.2). Hingegen kann man die Mechanismen der Fehlerbehebung (FAPMs *SReject* und *RReject*) nicht vollständig aus der XTP-Beschreibung herleiten. Der Grund dafür liegt in der Mißverständlichkeit der XTP-Beschreibung, die zu fehlerhaften Implementierungen führen kann. Abschnitt 6.5.3 beschreibt diese Fehlerquelle genauer. Die Methoden der Flußkontrolle (FAPMs *SCredit* und *RCredit*) entsprechen dagegen den Vorgaben (vgl. [XTP95], Abschnitt 4.4) ebenso wie diejenigen zur Auslösung von Meldungen durch den Empfänger (FAPMs *STriggerAck*, *RTriggerAck*, *STriggerCredit* und *RTriggerCredit*; vgl. [XTP95], Kapitel 3 und Abschnitt 4.6.2). Die Funktionalität der FAPMs *SCapacity*, *SSendInOrder*, *RGaps*, *RDuplicates*, *RReorderings*, *RPhantoms* und *RVoluntaryGaps* wird in der XTP-Beschreibung nicht explizit behandelt. Wir nehmen jedoch an, daß bei vollem Sendepuffer die Sendung neuer Dateneinheiten abgebrochen wird, und daß die Sendung und Auslieferung der Nutzdaten reihenfolgetreu erfolgt, wenn keine Lücken im Datenstrom aufgetreten sind.

Die Überprüfung der Konformität der Signalisierungskanäle ist für den Kanaltyp `"Active"` recht einfach, da das Vorgehen zur Aushandlung neuer Übertragungsraten und Datentransfermodi in [XTP95], Abschnitt 4.2.5 ausführlich erläutert ist. Der Abbau eines Simplex-Transferkanals wird in [XTP95], Abschnitt 4.2.7 beschrieben. Diese Beschreibungen sind zu den Spezifikationen der Schlüsselverwaltung (FAPMs *SBufferKey*, *RBufferKey*, *SSynchronize*), des Transports der Signalisierungsdaten (FAPMs *SBufferUsd*, *RBufferUsd*), der PDU-Bestätigungen (FAPMs *SAcknowledge*, *RAcknowledge*, *SShowDelivery*) und der Reihenfolgesicherung (FAPMs *SSendInOrder*, *RDuplicates*, *RReorderings*) konform. Die Transferkom-

[15] Nur das SC *DelayedClose* wird von mehreren FAPMs realisiert.

ponenten beim Aufbau einer Verbindung sind für Fehlersituationen nur kurz beschrieben. Aus [XTP95], Abschnitt 4.2.1 geht lediglich hervor, daß PDUs des Formats `FIRST` wiederholt werden. Da explizite Wiederholungsanforderungen (siehe Abschnitt 2.2.2) beim Verbindungsaufbau nicht sinnvoll sind — beim Verlust einer PDU hat der Empfänger keine Information über den Verbindungsaufbauwunsch und kann deshalb nicht reagieren — unterstellen wir, daß der Verbindungsaufbauinitiator PDUs des Formats `FIRST` wiederholt, wenn er über einen bestimmten Zeitraum keine Bestätigung des Verbindungsaufbaus erhält (Positive Acknowledgment with Retransmission, PAR). Da der Verbindungsaufbau implizit ist, und somit keine expliziten Bestätigungsnachrichten kennt, fassen wir alle PDUs vom Partner zum Initiator, die im Rahmen der Verbindung gesendet werden, als Bestätigung des Verbindungsaufbaus auf. Die FAPMs des Kanaltyps `"First"` wurden deshalb nach dieser Methode entworfen. Auch das Auftreten von Fehlern beim endgültigen Abbau einer Verbindung wird nur grob angedeutet (vgl. [XTP95], Abschnitt 4.2.7). Es wird festgestellt, daß PDUs mit gesetztem Steuerflag `END` nur einmal gesendet werden und der Sender danach für einen bestimmten Zeitraum in einen sogenannten „Zombie"-Zustand übergeht, in dem er auf Fehlermeldungen des Partners noch reagieren kann. Da die Art der Fehlermeldung und der Reaktion darauf nicht genauer erläutert wird, gehen wir bei der Entwicklung der FAPMs des Kanaltyps `"Last"` davon aus, daß jede Nachricht vom Verbindungspartner nach Sendung der Verbindungsabbau-PDU als Fehlermeldung gilt, worauf die Verbindungsabbau-PDU erneut gesendet wird. Die Möglichkeit von Verbindungsabbrüchen aufgrund von Inaktivität werden in [XTP95], Abschnitt 4.2.7 erwähnt. In diesem Fall werden PDUs vom Format `DIAG`, die für protokollinterne Verhandlungen vorgesehen sind, gesendet. Das Steuerflag `END` ist in diesen PDUs gesetzt. Die Fehlerbehebung ist zu der in Kanaltyp `"Last"` beschriebenen analog.

Die FAPMs der Intraconnection-Coordination beschreiben im wesentlichen den Aufbau und Abbau der Verbindung sowie die Aushandlung neuer Datentransfermodi. Das Verhalten der Protokollinstanzen beim Aufbau und Abbau von Verbindungen wird in [XTP95], Abschnitt 4.1 erörtert. Diese Beschreibung bildet die Grundlage für die Definition des vom Dienstnutzer geforderten Verhaltens in FAPM *UserBehaviour*[16]. Es berücksichtigt die in [XTP95], Abschnitte 4.2.1 und 4.2.7 erläuterten Verbindungsaufbau- bzw. Verbindungsabbaumodi und läßt die Neuaushandlung von Datenraten nur während des Datentransfers zu ([XTP95], Abschnitt 4.2.5). Diese Anforderungen werden auch von den anderen FAPMs der Intraconnection-Coordination (*ConnectionState*, *DataChanOpenW*, *DataChanOpenR*, *RDeliverBuff* und *RStopOld*) erfüllt. Die FAPMs *SMarkFinal* und *RMarkFinal* sind bis auf den in Abschnitt 6.5.3 vorgestellten Fehler ebenfalls zu der XTP-Beschreibung konform.

Die Vergabe eindeutiger Schlüssel wird in [XTP95], Abschnitte 4.2.2, 4.2.3 und 4.2.4 beschrieben. Diese Mechanismen werden bei der Entwicklung der FAPMs der Interconnection-Coordination berücksichtigt. Die Kennungsverwaltung in XTP entspricht der in den FAPMs *AssignCNo*, *SelectCNo*, *FreeCNo* und *LiveFreeCNo* spezifizierten Methode. Die Beschreibung der Koordinierung unterschiedlicher Verbindungen wird auch von den FAPMs *OpenConnections* und *FindDestination* eingehalten.

Wir können zusammenfassend feststellen, daß mit Ausnahme der im folgenden aufgeführ-

[16] In den FAPMs haben wir auf die explizite Spezifikation des Dienstnutzerkommandos `LISTEN` verzichtet, mit denen Nutzer vor dem Aufbau von Verbindungen Dienstgüteparameter angeben können, die sie zu tolerieren gewillt sind. Die Spezifikation bildet dieses Konstrukt dadurch ab, daß ein Verbindungsabbruch bei Verbindungsaufbauversuchen ohne akzeptable Dienstgüte vor der Auslieferung der Verbindungsaufbaunachricht erzeugt wird.

ten Unklarheiten Konformität zwischen den FAPMs und der im wesentlichen informellen XTP-Beschreibung besteht.

6.5.3 Unterspezifikation der Fehlerbehebung von XTP 4.0

Aus dem vorhergehenden Abschnitt wurde deutlich, daß die Beschreibung von XTP [XTP95] nicht für alle Protokollmechanismen eine präzise Beschreibung der jeweiligen Funktionalität enthält. Häufig kann man aus der vorhandenen Information mit etwas „Common Sense“ folgern, daß ein Mechanismus nur auf eine einzige Weise sinnvoll aufgebaut sein kann. Bei zumindest einem Protokollmechanismus können wir jedoch zeigen, daß man aus der Beschreibung von XTP zwei unterschiedliche Lösungen entwickeln kann, die für sich betrachtet beide sinnvoll sind, im Zusammenspiel miteinander jedoch zu groben Protokollfehlern führen.

Bei dem betroffenen Protokollmechanismus handelt es sich um die Erkennung und Meldung verlorengegangener Datenpakete unter Verwendung selektiver Wiederholung. Das Problem tritt auf, wenn das letzte Datenpaket vor dem Ende einer Verbindung oder der Neuaushandlung von Datenraten verlorengeht. Beim Empfang der PDU zum Verbindungsabbau bzw. zur Neuaushandlung der Datenrate bemerkt die Empfangsinstanz zwar, daß Daten verlorengegangen sind[17]. Jedoch wird aus der XTP-Beschreibung nicht deutlich, wie sie auf diese Störung des Ablaufs reagieren muß.

Zur Meldung von Übertragungsfehlern werden PDUs vom Format `ECNTL` (siehe Abschnitt 6.1.3) verwendet (vgl. [XTP95], Abschnitte 2.3 und 4.6.2). In diesen PDUs werden außer der Sequenznummer `rseq` der ersten verlorengegangenen Dateneinheit Spans, Bereiche korrekt empfangener Dateneinheiten, mit höheren Sequenznummern als `rseq` übermittelt. Der Sender überträgt danach erneut alle Dateneinheiten, deren Sequenznummer größer als `rseq` ist und die nicht in einem Span liegen. Das Problem ergibt sich nun daraus, daß eine PDU vom Typ `ECNTL` mindestens einen Span enthalten muß (siehe [XTP95], Abschnitt 2.3.2.1). In dem Fall, daß nur die zuletzt gesendete Daten-PDU verloren geht, gibt es jedoch keine korrekt empfangenen Dateneinheiten mit Sequenznummern größer als `rseq`, aus denen man einen Span bilden könnte. Unter Betrachtung der Konzepte von XTP kann man zu den folgenden beiden Lösungen kommen:

1. Man verwendet das Verfahren für Fehlerbehebung durch Zurücksetzung (go-back-n; siehe [XTP95], Abschnitt 2.3.2.2) und sendet eine PDU vom Format `ECNTL` mit der Sequenznummer der ersten verlorengegangenen Dateneinheit in `rseq` und einem Span, dessen beide Grenzwerte den Nachfolger der zuletzt gesendeten Dateneinheit referenzieren[18]. Für diese Lösung spricht, daß in dem betrachteten Fall die Fehlerbehebung der selektiven Zurückweisung mit der der Zurücksetzung identisch ist und deshalb die Verwendung des entsprechenden Mechanismus als natürlich erscheint.

2. Anstatt einer PDU vom Format `ECNTL` sendet der Empfänger der Daten ein Paket vom Format `TCNTL`, in dessen Feld `rseq` er die Sequenznummer der zuerst verlorengegangenen Dateneinheit angibt. Dieses Vorgehen basiert auf der Tatsache, daß die Neuaushandlung einer Datenrate oder ein Verbindungsabbau immer auch ein Synchronisations-Handshake ist (siehe [XTP95], Abschnitt 4.6.3), mit dem die beiden Stationen einer

[17] In Feld `seq` des PDU-Kopfs ist die Sequenznummer der zuletzt gesendeten Dateneinheit enthalten.

[18] Die Sequenznummer der zuletzt gesendeten Dateneinheit ist in einem Paket zum Verbindungsabbau bzw. zur Aushandlung der Datenrate enthalten.

Verbindung ihren Systemzustand aneinander anpassen. In diesem Fall führt das zu einer Rücksetzung des Senders in einen Zustand vor dem Ende des Datentransfers, der daraufhin die verlorengegangenen Dateneinheiten erneut sendet.

Beide Lösungen können also sinnvoll begründet werden. Wenn jedoch der Implementierer des Senders die Lösung 1 verwendet und die Wiederholung von Daten nur nach Eintreffen von PDUs des Formats `ENCTL` erlaubt, während der Empfänger die Lösung 2 unterstützt und keine derartige PDU sendet, kann ein schwerwiegender Protokollfehler auftreten. In Abhängigkeit von der Realisierung des Senders werden die Datenverluste entweder nicht erkannt, oder der Empfänger verhält sich aus der Sicht des Senders passiv, worauf dieser die Verbindung irgendwann abbricht.

6.5.4 Livenessverletzungen bei XTP

Abschließend müssen wir noch überprüfen, ob XTP Mechanismen besitzt, die die in den Liveness-FAPMs spezifizierten Fairnesseigenschaften verletzen. Wir nehmen an, daß Implementierungen Scheduler verwenden, die alle mit Fairnessanforderungen versehenen Aktionen fair behandeln. Jedoch kennt XTP einige Mechanismen, die bestimmte Funktionen so stark unterdrücken, daß deren Ausführung nicht mehr gewährleistet ist. Ein solcher Mechanismus ist das Prioritätskonzept von XTP. Jede PDU besitzt im Paketkopf ein Feld `SORT`, durch dessen Wert die Bearbeitungsreihenfolge bei Sendungen festgelegt wird (vgl. [XTP95], Abschnitt 2.2.5). Hochpriore PDUs werden stets bevorzugt, so daß deren zahlreiches Auftreten die Sendung niederpriorer PDUs verhindern kann, wodurch die Livenessanforderung in FAPM *SLiveMRq* verletzt wird. Lösen kann man dieses Problem durch Verwendung eines Schedulers, der hochpriore PDUs nur bevorzugt, wenn dadurch die Sendung der niederprioren PDUs nicht andauernd unterdrückt wird.

Eine weitere Quelle für Livenessverletzungen liegt im Mechanismus zur Initiierung von Meldungen. Der XTP-Paketkopf kennt die zwei Flags `SREQ` und `DREQ` zur Anforderung von Meldungen. `DREQ` verlangt, daß eine Meldung erst erfolgt, wenn die im aktuellen Paket befindlichen Nutzdaten und deren Vorgänger an den Dienstnutzer ausgeliefert sind. Dieses Verfahren garantiert die Fairness sowohl von Meldungen (FAPM *RLiveARq*) als auch von Auslieferungen (FAPM *RLiveIn*). Hingegen muß eine Meldung sofort erfolgen, wenn das Flag `SREQ` gesetzt ist. Der Grund dafür liegt darin, daß der Sender mit Hilfe dieser Methode Paketumlaufzeiten errechnen will. Für die Liveness der Auslieferungen besteht nun die Gefahr, daß bei häufiger Übertragung von PDUs mit gesetztem Flag `SREQ` vom Empfänger ständig Meldungen abgesendet werden müssen. In diesem Fall ist der Empfänger nicht mehr in der Lage, empfangene Dateneinheiten an den Dienstnehmer auszuliefern. Man muß also die Sendung von PDUs mit gesetztem Flag `SREQ` beschränken, so daß die Empfängerstation in der Lage ist, zwischen Meldungen auch Auslieferungen von Dateneinheiten durchzuführen. Die Lösungsvorschläge zu diesen Problemen sind nicht in der XTP-Beschreibung [XTP95] enthalten.

6.6 Einige Gedanken zur Anwendung des kompositionalen Stils

Nachdem wir den Einsatz des kompositionalen Spezifikationsstils für XTP dargestellt haben, möchten wir noch einmal kurz auf die Vorteile dieser Methode eingehen.

Als wesentlich erachten wir insbesondere, daß der kompositionale Spezifikationsstil ein Korsett zur Reduktion eines komplexen Protokolls wie XTP bietet. Bei der Parametrisierung eines FAPMs mußte immer nur ein kleiner Teil der gesamten Funktionalität von XTP erarbeitet werden. Dadurch wird das Verständnis des Zusammenspiels der einzelnen Protokollmechanismen gefördert. So fiel es uns zum Beispiel leicht, die in den Abschnitten 6.5.3 und 6.5.4 beschriebenen Fehlerquellen zu entdecken.

Weiterhin konnten wir unter Verwendung des Frameworks sehr einfach durch Parametrisierung eine formale, modulare Spezifikation von XTP erzeugen. Die Arbeitszeit für die Parametrisierung betrug etwa eine Woche. Die erzeugten Spezifikationen der XTP-Protokollmechanismen sind hinreichend abstrakt, um Implementierungsvorentscheidungen auszuschließen. Sie verlangen jedoch im Gegensatz zu Beschreibungen in natürlicher Sprache die Modellierung aller für den Protokollablauf wesentlichen Komponenten. Unvollständige Protokollbeschreibungen wie [XTP95] werden dadurch ausgeschlossen. Die Spezifikationsbausteine können durch Komposition einfach und automatisch (vgl. [Hey95]) in Spezifikationen des Gesamtprotokolls umgeformt werden. Außerdem kann man sie in bekanntere Beschreibungsformate wie State Charts oder Zustandsübergangsautomaten überführen. Zum Beispiel kann man Spezifikationen mit dem in [Kru90a, Kru90b] vorgeschlagenen Verfahren in strukturierte Erreichbarkeitsgraphen überführen. Die einzelnen Erreichbarkeitsgraphen beschreiben dann Abstraktionen der Spezifikationen in Form von Zustandsübergangsautomaten. Derartige textuelle oder graphische Beschreibungen eignen sich auch für eine „offizielle“ Protokollbeschreibung, weil sie zum einen präziser als die natürliche Sprache sind. Zum anderen fördert die Modularität der Beschreibung auch das Verständnis der Funktionalität einzelner Protokollmechanismen und ihrer Kooperation.

Ein weiterer Vorteil liegt in der Qualitätssicherung. Auf der Basis der formalen Spezifikationen von Protokollmechanismen und Diensteigenschaften konnte unter Verwendung der Theoreme des Spezifikationsstils relativ einfach nachgewiesen werden, welche Diensteigenschaften von der XTP-Spezifikation erfüllt werden. Die Durchführung der Beweise unter Verwendung der Theoreme des Frameworks benötigte ebenfalls ungefähr eine Woche.

Schließlich soll noch auf den Aspekt der Wiederverwendbarkeit von Spezifikationen eingegangen werden. Mit Ausnahme des FAPMs *SSynchronize* und der zugehörigen SCs *SDUIdSynch*, *CorruptionsSynch*, *CapacitySynch* und *ShowDeliverySynch* sind alle Spezifikationen der Protokollfunktionen und Diensteigenschaften Instantiierungen von Frameworkbausteinen. *SSynchronize*, die oben genannten SCs und die zugehörigen Theoreme mußten zwar neu spezifiziert bzw. verifiziert[19] werden. Sie werden dem Framework hinzugefügt und können bei späteren Anwendungen eingesetzt werden.

Der Zeitraum für die Erstellung der gesamten Spezifikation und Verifikation von XTP einschließlich Dokumentation betrug insgesamt 18 Tage.

[19] Die Verifikation dieser Theoreme wurde durch die im Framework vorhandenen Theoreme vereinfacht (siehe Abschnitt 6.3.3).

Kapitel 7

Schlußbetrachtungen

Die Spezifikation und Verifikation von XTP zeigte, daß unser Ziel erreicht wurde, den Aufwand einer auf formalen Methoden basierten Entwicklung von Hochleistungstransportprotokollen zu senken. Es stellt sich allerdings noch die Frage nach der Vollständigkeit der Spezifikationsbaustein- und Theorembibliotheken. Das Framework wird von Protokollentwicklern nur akzeptiert, wenn sie die Spezifikationen zumindest fast aller Protokollmechanismen und Diensteigenschaften durch Instantiierung vorhandener FAPMs bzw. SCs erzeugen können. Das Framework kann zwar einfach um neue Bausteine und Theoreme erweitert werden (wie mit dem FAPM *SSynchronize* gezeigt). Jedoch sollte die Notwendigkeit, daß der Spezifizierer selbst neue Frameworkkomponenten erzeugt und verifiziert, nicht der Regelfall sein. Wir haben neben XTP die meisten anderen Transportprotokolle auf vollständige Spezifizierbarkeit durch das Framework überprüft. Die klassischen Transportprotokolle TCP und ISO/OSI-TP4 wie auch die Hochleistungstransportprotokolle NETBLT, VMTP, SNR und PATROCLOS verwenden nur Protokollmechanismen, die auch in XTP enthalten sind, und können deshalb durch die um *SSynchronize* erweiterten FAPMs des Frameworks spezifiziert werden. Dagegen erlaubt MSP im Gegensatz zu XTP dem Dienstnehmer, gleichzeitig Daten über sieben unterschiedliche virtuelle Datenkanäle mit jeweils unterschiedlicher Dienstgüte zu übertragen. Wir können jedoch auch dieses Protokoll mit den vorhandenen FAPMs des Frameworks spezifizieren, indem wir für jeden Kanal eine eigene Transferkanalspezifikation erzeugen. Die Prozesse der Intraconnection-Coordination werden so ausgelegt, daß am Anfang der Verbindung alle sieben Transferkanäle aktiv werden und bis zum Verbindungsende aktiv bleiben.

Wir möchten nun noch kurz skizzieren, wie man das vorgestellte Framework und den dazugehörenden kompositionalen Spezifikationsstil weiterentwickeln kann. Als notwendig erscheint uns die Einführung von Bausteinen, mit denen man Mehrbenutzerverbindungen modellieren kann. Der Bedarf derartiger Verbindungen (zum Beispiel Videokonferenzen) steigt rapide an. Wir favorisieren eine Struktur, in der eine Verbindung zwischen n Benutzern durch $n \cdot (n-1)$ Monocast-Datenkanäle modelliert wird, die jeweils zwei Teilnehmer miteinander verbinden. Der Vorteil dieser Anordnung ist, daß die Erweiterungen nur die FAPMs der Intraconnection-Coordination betreffen. Diese FAPMs spezifizieren die Verteilung der Daten auf die verschiedenen Datenkanäle sowie die Verwaltung und Moderation der Multicast-Verbindungen. Diese Erweiterung wird im Rahmen einer Diplomarbeit vorgenommen.

Ein weiterer Ansatzpunkt liegt in der Unterstützung des Protokollentwicklers bei der Auswahl von Theoremen aus dem Framework. Im Rahmen einer Diplomarbeit [Drö96] wurde

ein Werkzeug entwickelt, das als Eingabe je eine kompositionale cTLA$^+$-Spezifikation eines Protokolls und eines Dienstes enthält. Es durchsucht die Theorembibliothek nach geeigneten Theoremen und überprüft deren Randbedingungen. Die Parameterbedingung `Pars` eines Theorems wird in die Eingabesprache pTLA$^+$ des Theorembeweisers OTTER übersetzt (vgl. [Gei94]). Alle anderen Randbedingungen der Theoreme werden durch das Werkzeug selbständig überprüft.

Außerdem kann man den Frameworkansatz auch in anderen Anwendungsgebieten einsetzen. Im Rahmen des Projektes[1] „Modulare temporallogische Spezifikationen hybrider Systeme“ entwickeln wir ein Framework zum Entwurf von Steuerungen kontinuierlich-diskreter technischer Systeme (vgl. [HK97b, HK97c]). Dazu muß der Spezifikationsstil um Konstrukte zur Modellierung von Realzeit erweitert werden. Wir verwenden eine Methode, die auf dem von Abadi und Lamport in [AL91b] vorgeschlagenen Verfahren beruht. Die Zeit wird dort durch eine virtuelle Uhr modelliert. Eine Anforderung, daß eine Aktion π frühestens nach einer bestimmten Zeit schalten darf, wird in der Schaltbedingung $\mathit{Enabled}(\pi)$ modelliert. Dagegen spezifiziert man, daß π spätestens nach einer bestimmten Zeit t geschaltet hat, durch eine Bedingung an die Uhr. Diese darf den Wert t erst überschreiten, wenn π geschaltet hat. Diese Modellierung birgt die Gefahr des sogenannten Zeno-Verhaltens, in dem ein Zeitpunkt t niemals überschritten wird. Um die Spezifikation derartiger de facto nicht erfüllbarer Systeme auszuschließen, verwendet der erweiterte kompositionale Stil eine zu der Abschwächung der Fairnessanforderungen in den Prozessen (durch Umgebungsbereitmengenvariablen) analoge Methode. Die Bedingung, daß eine Aktion π spätestens zum Zeitpunkt t geschaltet haben muß, wird nur eingehalten, wenn alle miteinander komponierten Prozesse die Schaltung von π zu einem Zeitpunkt vor t gestatten. Dadurch wird ein Zeno-Verhalten ausgeschlossen. Die Garantie, daß π garantiert vor dem Zeitpunkt t geschaltet wird, kann man durch Einhaltung spezieller Stilkonventionen sicherstellen.

Schließlich stellen wir noch eine andere Anwendungsmöglichkeit des kompositionalen Spezifikationstils vor. Dieser Ansatz [MK95] sieht die Konstruktion von verteilten Ortsverbundsanwendungen durch Integration von Entwurfsmustern [GHJV93] unter Verwendung korrektheitserhaltender Prozeßtransformationen (vgl. [Bol92, DW92, KSJ88]) vor. Dem Systementwickler steht eine Bibliothek von Entwurfsmustern zur Verfügung. Er entwirft zunächst sein Anwendungsproblem ohne Berücksichtigung der Verteilungsaspekte. In einem zweiten Schritt wird die Verteilung durch cTLA$^+$-Prozeßintegration von Spezifikationen verteilter Kontrollalgorithmen eingeführt. Die noch relativ abstrakten Kontrollalgorithmen werden in einem weiteren Schritt durch Integration von Spezifikationen von ISO/OSI-Anwendungsdiensten konkretisiert. In einem vierten Schritt zerlegt man das verteilte System durch cTLA$^+$-Prozeßzerlegung in lokale Komponenten, die implementiert werden können.

[1] Dieses Projekt wird von der Deutschen Forschungsgemeinschaft im Rahmen des Schwerpunktprogramms „KONDISK“ gefördert.

Literaturverzeichnis

[ACD91] Rajeev Alur, Costas Courcoubetis und David Dill. Model-checking for Probabilistic Real-Time Systems (extended abstract). Aus J. Leach Albert, B. Rodriguez Artalejo und B. Monien, Hrsg., *Automata, Languages and Programming*, Nummer 510 in Lecture Notes in Computer Science, Madrid, Spanien, Juli 1991. Springer-Verlag, New York, NY.

[AK89] A. S. Acampora und M. J. Karol. An overview of lightwave packet networks. *IEEE Network*, 3(1):29–41, Januar 1989.

[AL91a] Martín Abadi und Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, Mai 1991.

[AL91b] Martín Abadi und Leslie Lamport. An Old-Fashioned Recipe for Real Time. Aus J. W. de Bakker, C. Huizing, W. P. de Roever und G. Rozenberg, Hrsg., *Real-Time: Theory in Practice*, Nummer 600 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[AL93a] Martín Abadi und Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Januar 1993.

[AL93b] Martín Abadi und Leslie Lamport. Conjoining Specifications. Interner Bericht 118, DEC Digital Systems Research Center, Palo Alto, Dezember 1993.

[ANS92] ANSI. *American National Standard X3.139; Fiber Distributed Data Interface (FDDI) — Token Ring Media Access Control (MAC)*, 5. Ausgabe, 1992.

[AO91a] Krzysztof R. Apt und Ernst-Rüdiger Olderog. Introduction to Program Verification. Aus Erich J. Neuhold und Manfred Paul, Hrsg., *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, Seiten 363–429. Springer-Verlag, Berlin, 1991.

[AO91b] Krzysztof R. Apt und Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1991.

[AP93] Mark B. Abbott und Larry L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE / ACM Transactions on Networking*, 1(5), Oktober 1993.

[AS85] Bowen Alpern und Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, 1985.

[AW91] Henrik Reif Andersen und Glynn Winskel. Compositional Checking of Satisfaction. Aus Larsen und Skou [LS91c], Seiten 24–36.

[Bar85] D. Barstow. Automatic programming for streams. Aus *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Seiten 232–237, 1985.

[Bar88] D. Barstow. Automatic programming for streams II. Aus *Proceedings of the 10th International Conference on Software Engineering*, Seiten 439–447, 1988.

[BC95] Dieter Barnard und Simon Crosby. The Specification and Verification of an Experimental ATM Signalling Protocol. Aus Piotr Dembiński und Marek Średniawa, Hrsg., *Protocol Specification, Testing, and Verification XV*, Seiten 155–170, Warschau, Polen, 1995. IFIP, Chapman & Hall.

[BCF93] E. Barbero, M. Cociglio und F. Ferrero. Evaluating high-speed networks and services: issues, preliminary results and future trends. *CSELT Technische Berichte*, 21(4):813–831, Oktober 1993.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill und H. L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDD+92] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner und Rainer Weber. The Design of Distributed Systems — An Introduction to FOCUS. Interner Bericht TUM-I9202, Technische Universität München, Institut für Informatik, Postfach 202420, D-8600 München, Januar 1992.

[BE91] E. Best und J. Esparza. Model checking of persistent Petri nets. Aus E. Börger, Hrsg., *Computer Science Logic' 91, 5th Workshop*, Seiten 35–52, Bern, Switzerland, 1991. Springer-Verlag Berlin.

[Bil91] Jonathan Billington. Formal Specification of Protocols: Protocol Engineering. *Encyclopedia of Microcomputers*, 7:299–314, Marcel Dekker, NY, USA 1991.

[Bla94] Holger Blaskowski. Theorembeweiser-gestützte Verifikation von Fairness-Eigenschaften verteilter Systeme. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, Juli 1994.

[BLB93] Ed Brinksma, Rom Langerak und Peter Broekroelofs. Functionality Decomposition by Compositional Correctness Preserving Transformation. Aus Costas Courcoubetis, Hrsg., *Verification*, Nummer 697 in Lecture Notes in Computer Science, Elounda, Griechenland, Juni 1993. Springer Verlag Berlin, Heidelberg, New York.

[Bol92] T. Bolognesi (Hrsg.). Catalogue of LOTOS Correctness Preserving Transformations. Final Report. Interner Bericht Deliverable Lo/WP1/TI.2/N0045/V03, ESPRIT Project 2304 (Lotosphere), Januar 1992.

[BOP87] H. J. Burkhardt, P. Ochsenschläger und R. Prinoth. Produktnetze — Ein formales Beschreibungsmittel für kooperierende Systeme. GMD – Studien 129, GMD, 1987.

[BR92] Torsten Braun und Ortwin Rose. Hochgeschwindigkeits-Transportprotokolle — Architekturen und Mechanismen. *Praxis der Informationsverarbeitung und Kommunikation*, 15(3):136–144, Juli 1992.

[Bra93] Torsten Braun. *Ein paralleles Transportsubsystem für zellenbasierte Hochgeschwindigkeitsnetze.* Doktorarbeit, Universität Karlsruhe, 1993. VDI-Fortschrittbericht Nr. 264, VDI-Verlag.

[Bri90] Ed Brinksma. Constraint-oriented specification in a constructive formal description technique. Aus Jaco. W. de Bakker, Willem Paul de Roever und Grzegorz Rozenberg, Hrsg., *Stepwise Refinement of Distributed Systems*, Nummer 430 in Lecture Notes in Computer Science, Seiten 130–152. Springer Verlag, 1990.

[Bro92] Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. Interner Bericht 75, DEC Digital Systems Research Center, Palo Alto, Februar 1992. Forschungsbericht.

[BSZ92] Torsten Braun, Burkhard Stiller und Martina Zitterbart. XTP and VMTP on Multiprocessor Architectures. Aus *Proceedings of the International Workshop on Advanced Communications and Applications for High-Speed Networks*, Seiten 67–76, München, März 1992.

[Bud92] Stan Budkowski. Estelle Development Toolset. *Computer Networks and ISDN Systems Journal*, 25(1):63–82, 1992. Special Issue on FDT Concepts and Tools.

[BvdM93] James A. Brown und Mike van der Meulen. APL support off parallel and cluster architectures. Aus *Proceedings SHARE Europe Spring Meeting. Distributed Applications*, Seiten 83–93, Carouge/Genf, Schweiz, 1993. SHARE Europe.

[BWH86] J. Billington und M. Wilbur-Ham. Automated Protocol Verification. Aus *Protocol Specification, Testing, and Verification V*, IFIP, Elsevier 1986.

[BZ83] D. Brand und P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–343, April 1983.

[BZ93] Torsten Braun und Martina Zitterbart. Parallel Transport System Design. Aus A. Danthine und O. Spaniol, Hrsg., *Proceedings of High Performance Networking, IV*, Seiten 397–412. IFIP, North-Holland, 1993.

[CA90] Y. T. Cheung und J. W. Atwood. Specifying the Xpress transfer protocol using Estelle and Valira. Aus J. Quemada, J. Manas und E. Vazquez, Hrsg., *Formal Description Techniques, III.*, Seiten 503–517, Madrid, Spanien, 1990. North-Holland.

[Cat95] Octavian Catrina. Protocol analysis and verification methods, application to the Xpress Transport Protocol 4.0. Aus Piotr Dembiński und Marek Średniawa, Hrsg., *Protocol Specification, Testing, and Verification XV*, Seiten 385–400, Warschau, Polen, 1995. IFIP, Chapman & Hall.

[CES83] E. M. Clarke, E. A. Emerson und A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. Aus *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, Seiten 117–126, Austin, 1983.

[CES86] E. M. Clarke, E. A. Emerson und A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions of Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGL94] E. M. Clarke, O. Grumberg und D. E. Long. Model Checking and Abstraction. *ACM Transactions of Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGS91] Costas Courcoubetis, Susanne Graf und Joseph Sifakis. An Algebra of Boolean Processes. Aus Larsen und Skou [LS91c], Seiten 454–465.

[Che86] David R. Cheriton. VMTP: A protocol for the next generation of communication systems. Aus *Proceedings of ACM SIGCOMM '86 Symposium, Stowe, VT., USA*, Seiten 406–415, August 1986.

[Che87] Greg. L. Chesson. The Protocol Engine Project. *UNIX Review*, 5(9):70–77, September 1987.

[Che88] David R. Cheriton. WMTP: Versatile Message Transaction Protocol — Protocol Specification. Request for Comments RFC 1045, Network Working Group, Februar 1988.

[Che91] Greg. L. Chesson. The Evolution of XTP. Aus André Danthine und Otto Spaniol, Hrsg., *Proceedings of 3rd IFIP Conference on High-Speed Networking*, Seiten 15–24, Berlin, März 1991. IFIP.

[Chi89] G. Chiola. Compiling techniques for the analysis of stochastic Petri Nets. Aus *4th International Conference on Modelling Techniques and Tools*, Mallorca, Spanien, 1989.

[CJRS89] David D. Clark, Van Jacobson, John Romkey und Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Network Magazine*, Seiten 23–29, Juni 1989.

[CK90] E. M. Clarke und R. P. Kurshan, Hrsg. *Computer-Aided Verification*, Nummer 531 in Lecture Notes in Computer Science, Berlin Heidelberg, 1990. Springer-Verlag.

[CLZ87] D. D. Clark, M. L. Lambert und L. Zhang. NETBLT: A Bulk Data Transfer Protocol. Interner Bericht RFC 998, Network Working Group, MIT, März 1987.

[CM88] K. Mani Chandy und Jayadev Misra. *Parallel Program Design — A Foundation.* Addison Wesley, 1988.

[CP95] S. Christensen und L. Petrucci. Modular state space analysis of coloured Petri nets. Aus *16th International Conference on Application and Theory of Petri Nets.*, Springer-Verlag, 1995.

[CR90] David A. Carrington und K. A. Robinson. Computer Assistance for Program Refinement. Aus Clarke und Kurshan [CK90], Seiten 312–321.

[CT90] David D. Clark und David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *Computer Communications Review*, 20(4):200–208, September 1990. ACM SIGCOMM '90 Symposium, Philadelphia.

[CVWY90] Costas Courcoubetis, Moshe Vardi, Pierre Wolper und Mihalis Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. Aus Clarke und Kurshan [CK90], Seiten 233–242.

[CW89] David R. Cheriton und Carey L. Williamson. VMTP as the Transport Layer for High-Performance Distributed Systems. *IEEE Communications Magazine*, 27:37–44, Juni 1989.

[DDK+90] Willibald A. Doeringer, Doug Dykeman, Matthias Kaiserswerth, Werner Meister, Harry Rudin und Robin Williamson. A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[DHWT91] David L. Dill, Alan J. Hu und Howard Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. Aus Larsen und Skou [LS91c], Seiten 255–265.

[Dij68] E. W. Dijkstra. The structure of "THE"-multiprogramming system. *Communications of the ACM*, 11:341–346, Mai 1968.

[dP94] Martin de Prycker. *Asynchronous Transfer Mode — Die Lösung für Breitband-ISDN*. Prentice Hall Verlag, München, 1994.

[Drö96] Olaf Drögehorn. Ein Werkzeug zum formal basierten Entwurf von Hochleistungstransportprotokollen. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, November 1996.

[DTH92] Sylvie Dupuy, Wassim Tawbi und Eric Horlait. Protocols for high-speed multimedia communications network. *Computer Communications*, 15(6):349–358, Juli 1992.

[DW92] Claus Dendorfer und Rainer Weber. From Service Specification to Protocol Entity Implementation — An Exercise in Formal Protocol Development. Aus R. J. Lynn, Jr. und M. Ü. Uyar, Hrsg., *Protocol Specification, Testing, and Verification XII*, Seiten 163–178. Elsevier, 1992.

[EGL92] Urban Engberg, Peter Grønning und Leslie Lamport. Mechanical Verification of Concurrent Systems with TLA. Aus v. Bochmann, G. und D. K. Probst, Hrsg., *Verification*, Nummer 663 in Lecture Notes in Computer Science, Seiten 44–55, 4th International Workshop, CAV '92, Montreal, 1992. Springer-Verlag.

[EH86] E. Allen Emerson und Joseph Y. Halpern. "Sometimes" and "Not Never" revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, Januar 1986.

[EHK93] Ulrich Eickhoff, Peter Herrmann und Heiko Krumm. Theorem Prover Based Verification of TLA Systems. Forschungsbericht 492, Universität Dortmund, D-44221 Dortmund, Mai 1993.

[Eic92] Ulrich Eickhoff. Rechnergestützte Verifikation verteilter Systeme unter Einsatz von Beweiswerkzeugen. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, September 1992.

[Eil74] S. Eilenberg. *Automata, Languages and Machines*, Jahrgang A. Academic Press, New York, 1974.

[EM85] H. Ehrig und B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[Fai85] Richard Fairley. *Software Engineering Concepts.* McGraw-Hill, 1985.

[Fel90] David C. Feldmeier. Multiplexing Issues in Communication System Design. *Computer Communications Review*, 20(4):209–219, September 1990.

[Fel93] David C. Feldmeier. A Framework of Architectural Concepts for High-Speed Communication Systems. *IEEE Journal on Selected Areas in Communications*, 11(4):480–488, Mai 1993.

[FM91] Jean-Claude Fernandez und Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. Aus Larsen und Skou [LS91c], Seiten 181–191.

[Fra83] A. G. Fraser. Towards a Universal Data Transport System. *IEEE Journal on Selected Areas in Communications*, 1:803–816, November 1983.

[Fra86] Nissim Francez. *Fairness.* Springer-Verlag, New York, 1986.

[Fre87] Johannes Freudenmann. Development of Communication Software by Stepwise Refinement. Aus *Protocol Specification, Testing and Verification VII*, Seiten 391–404, 1987.

[Gei93] Michael Geikowski. Werkzeugunterstütztes Einarbeiten in Spezifikationen verteilter Systeme in TLA+. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, Juli 1993.

[Gei94] Ansgar Geist. Eine Theorembeweiser-gestützte Entwicklungsumgebung für die halbautomatische Verifikation verteilter Systeme. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, Dezember 1994.

[GGH90] Stephen J. Garland, John V. Guttag und James J. Horning. Debugging Larch Shared Language Specifications. *IEEE Transactions of Software Engineering*, 16(9):1044–1057, September 1990.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. Aus O. M. Nierstrasz, Hrsg., *7th European conference on Object oriented programming*, Kaiserslautern, 1993. Springer Verlag.

[Gib94] W. Wayt Gibbs. Software's Chronic Crisis. *Scientific American*, 271(3):72–81, September 1994.

[GKWW89] D. Giarrizzo, M. Kaiserswerth, T. Wicki und R. C. Williamson. High-Speed Parallel Protocol Implementation. Aus H. Rudin und R. Williamson, Hrsg., *Proceedings of IFIP Workshop on Protocols for High-Speed Networks*, Zürich, Schweiz, Mai 1989. IFIP.

[God90] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. Aus Clarke und Kurshan [CK90], Seiten 176–185.

[Gol90] David M. Goldschlag. Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1023, September 1990.

[GP93] P. Godefroid und D. Pirottin. Refining dependencies improves partial-order verification methods. Aus *5th International Conference on Computer Aided Verification*, Nummer 697 in Lecture Notes in Computer Science, Springer-Verlag, 1993.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi und Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. Aus Piotr Dembiński und Marek Średniawa, Hrsg., *Protocol Specification, Testing, and Verification XV*, Seiten 3–18, Warschau, Polen, 1995. IFIP, Chapman & Hall.

[Gra93] Günther Graw. Interpretationen von Spezifikationen verteilter Systeme in TLA+. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, September 1993.

[Gre91] Paul E. Green. The Future of Fiber-Optic Computer Networks. *Computer*, 24:78–87, September 1991.

[Haa91] Zygmund Haas. A Protocol Structure for High-Speed Communication over Broadband ISDN. *IEEE Network Magazine*, Seiten 64–70, Januar 1991.

[Hal92] Fred Halsall. *Data communications, computer networks, and open systems.* Addison Wesley, 3. Ausgabe, 1992.

[Hei92] Bernd Heinrichs. XTP Specification and Parallel Implementation. Aus *Proceedings of the International Workshop on Advanced Communications and Applications for High Speed Networks*, Seiten 77–84, München, 1992. Siemens.

[Her92] Peter Herrmann. Specification and Verification of the Sliding Window Protocol in TLA. Interner Bericht Dok-TLA92/14, Universität Dortmund, Informatik IV, D-44221 Dortmund, September 1992.

[Her93] Peter Herrmann. Verifikation von Sicherheits- und eingeschränkten Lebendigkeitseigenschaften. Interner Bericht Dok-L07-93/29, Universität Dortmund, Informatik IV, D-44221 Dortmund, Januar 1993.

[Hey95] Carsten Heyl. Werkzeugunterstützung des kompositionalen Entwurfs verteilter Systeme in TLA. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, 1995.

[HK90] Z. Har' El und R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Januar 1990.

[HK92] Peter Herrmann und Heiko Krumm. Report on Analysis and Verification Techniques. Forschungsbericht 485, Universität Dortmund, D-44221 Dortmund, November 1992.

[HK94] Peter Herrmann und Heiko Krumm. Compositional Specification and Verification of High-Speed Transfer Protocols. Aus Son T. Vuong und Samuel T. Chanson, Hrsg., *Protocol Specification, Testing, and Verification XIV*, Seiten 339–346, Vancouver, B.C., Kanada, 1994. IFIP, Chapman & Hall.

[HK95] Peter Herrmann und Heiko Krumm. Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations. Aus Piotr Dembiński und Marek Średniawa, Hrsg., *Protocol Specification, Testing, and Verification XV*, Seiten 171–186, Warschau, Polen, 1995. IFIP, Chapman & Hall.

[HK97a] Peter Herrmann und Heiko Krumm. Modular Specification and Verification of XTP. Aus *Proceedings of the 5th International Conference on Telecommunication Systems — Modeling and Analysis*, Seiten 477–486, Nashville, TN., USA, 1997. IFIP.

[HK97b] Peter Herrmann und Heiko Krumm. Specification of Hybrid Systems in cTLA+. Erscheint in *Proceedings of the 5th International Workshop on Parallel & Distributed Real-Time Systems — (WPDRTS'97)*, Genf, Schweiz, 1997. IEEE.

[HK97c] Peter Herrmann und Heiko Krumm. Kompositionale Constraints hybrider Systeme. Aus E. Schnieder und D. Abel, Hrsg., *Entwurf komplexer Automatisierungssysteme*, Seiten 243–264, Braunschweig, 1997.

[HKKS94] Peter Herrmann, Thomas Kraatz, Heiko Krumm und Miriam Stange. Automated Verification of Refinements of Concurrent and Distributed Systems. Forschungsbericht 541, Universität Dortmund, Informatik IV, D-44221 Dortmund, März 1994.

[HM85] Matthew Hennessy und Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, Januar 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[Hog89] Dieter Hogrefe. *Estelle, LOTOS und SDL.* Springer-Verlag, 1989.

[Hol87] Gerard J. Holzmann. On Limits and Possibilities of Automated Protocol Analysis. Aus H. Rudin und C. H. West, Hrsg., *Protocol Specification, Testing, and Verification VII*, Seiten 339–344. North-Holland, 1987.

[Hol88] Gerard J. Holzmann. Algorithms for Automated Protocol Verification. *Software — Practice and Experience*, 18(2):137–161, Februar 1988.

[Hol90] Gerard J. Holzmann. Algorithms for Automated Protocol Verification. *AT&T Technical Journal*, Seiten 32–44, Januar 1990.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hol93] Gerard J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25:981–1017, 1993.

[Hol94] Gerard J. Holzmann. The Theory and Practice of A Formal Method: NewCoRe. Aus B. Pehrson und I. Simon, Hrsg., *13th World Computer Congress 94*, Jahrgang I. IFIP, Elsevier, 1994.

[Hol95] Gerard J. Holzmann. An analysis of bitstate hashing. Aus Piotr Dembiński und Marek Średniawa, Hrsg., *Protocol Specification, Testing, and Verification XV*, Seiten 301–314, Warschau, Polen, 1995. IFIP, Chapman & Hall.

[HP91] N. C. Hutchinson und L. L. Peterson. The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17:64–76, Januar 1991.

[HS91] R. Herrtwich und R. Steinmetz. Towards Integrated Multimedia Systems: Why and How. Interner Bericht 43.91.01, IBM European Networking Center, Heidelberg, 1991.

[HU79] J. E. Hopcroft und J. D. Ullman. *Introduction To Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[HV91] Joseph Y. Halpern und Moshe Y. Vardi. Model Checking vs. Theorem Proving: a Manifesto. Aus Vladimir Lifschitz, Hrsg., *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Seiten 151–176. Academic Press, San Diego, CA, 1991.

[IEE90] IEEE, New York, NY, USA. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 610.12, 1990.

[IEE94] IEEE, New York, NY, USA. *802.6: Distributed Queue Dual Bus (DQDB) Subnetwork of a Metropolitan Area Network (MAN)*, 1994.

[IEE95] IEEE, New York, NY, USA. *802.5: Token Ring Access Method*, 1995.

[IEE96] IEEE, New York, NY, USA. *802.3: Carrier Sense Multiple Access with Collision Detection*, 1996.

[ISO86a] ISO. *Information Processing Systems — Open Systems Interconnection — Connection Oriented Transport Protocol Specification*, International Standard ISO/IS 8073, 1986.

[ISO86b] ISO. *Information Processing Systems — Open Systems Interconnection — Protocol for Providing the Connectionless-Mode Transport Service*, International Standard ISO/IS 8602, 1986.

[ISO86c] ISO. *Information Processing Systems — Open Systems Interconnection — Transport Service Definition*, International Standard ISO/IS 8072, 1986.

[ISO89a] ISO. *ESTELLE: A formal description technique based on an extended state transition model*, International Standard ISO/IS 9074, 1989.

[ISO89b] ISO. *LOTOS: Language for the temporal ordering specification of observational behaviour*, International Standard ISO/IS 8807, 1989.

[ISO94] ISO. *Information Processing Systems — Open Systems Interconnection — Basic Reference Model*, International Standard ISO/IS 7498, 1994.

[ITU92] ITU-T. *1992 Set of B-ISDN-I Series Recommendations*, Juni 1992.

[ITU93] ITU-T. *Recommendation Z.100: CCITT Specification and Description Language SDL*, 1993.

[Jac88] Van Jacobson. Congestion Avoidance and Control. Aus *Proceedings of ACM SIGCOMM'88 Symposium*, Seiten 314–329, Stanford, CA., USA, August 1988.

[JJ91] Claude Jard und Thierry Jéron. Bounded-memory Algorithms for Verification On-the-fly. Aus Larsen und Skou [LS91c], Seiten 192–202.

[JK90] Ryszard Janicki und Maciej Koutny. Using Optimal Simulations to Reduce Reachabilitiy Graphs. Aus Clarke und Kurshan [CK90], Seiten 166–175.

[JW78] Kathleen Jensen und Niklaus Wirth. *PASCAL — User Manual and Report.* Springer-Verlag, New York, 2. Ausgabe, 1978.

[KC88] Hermant Kanakia und David R. Cheriton. The VMP Network Adapter Board — High-Performance Network Communications for Multiprocessors. Aus *Proceedings of ACM SIGCOMM'88 Symposium*, Seiten 175–187, Stanford, CA., USA, August 1988.

[Kem96] Peter Kemper. Reachability Analysis Based on Structured Representation. Aus Jonathan Billington and Wolfgang Reisig, Hrsg., *Application and Theory of Petri Nets 1996*, Nummer 1091 in Lecture Notes in Computer Science, Springer-Verlag, 1996.

[Knu73] D. E. Knuth. *The art of computer programming*, Jahrgang 3. Addison-Wesley, Reading, Mass., 1973.

[KR90] Brian W. Kernighan und Dennis M. Ritchie. *Programmieren in C.* PC professionell. Hanser Verlag, 2. Ausgabe, 1990.

[Kru90a] Heiko Krumm. *Funktionelle Analyse von Kommunikationsprotokollen.* Nummer 247 in Informatik-Fachberichte. Springer-Verlag, Heidelberg, 1990.

[Kru90b] Heiko Krumm. Projections of the Reachability Graph and Environment Models. Aus J. Sifakis, Hrsg., *Automatic Verification Methods for Finite State Systems*, Nummer 407 in Lecture Notes in Computer Science, Berlin, 1990. Springer-Verlag.

[KSJ88] R. Kurki-Suonio und H.-M. Järvinen. Action system approach to the specification and design of distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(10):510–554, Oktober 1988.

[Kun92] H. T. Kung. Gigabit Local Area Networks: A Systems Perspective. *IEEE Communications Magazine*, Seiten 79–89, April 1992.

[Kus93] Michael Kuschke. Animation verteilter Systeme in TLA. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, Juli 1993.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[Lam80] Leslie Lamport. "Sometimes" is sometimes "not ever": A Tutorial on the temporal logic of programs. Aus *Proceedings of the Seventh Annual Symposium on Principles of Programming Languages (POPL)*, Seiten 174–185. ACM SIGACT–SIGPLAN, ACM, Januar 1980.

[Lam82] Leslie Lamport. Basic Concepts — Logical Foundation. Aus M. Paul und H. J. Siegert, Hrsg., *Distributed Systems*, Nummer 190 in Lecture Notes in Computer Science, Kapitel 2.2, Seiten 19–30. Springer-Verlag, Berlin, 1982.

[Lam89] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, Januar 1989.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, Mai 1994.

[Lam95] Leslie Lamport. TLA$^+$. Forschungsbericht, DEC Digital Systems Research Center, Palo Alto, Juli 1995.

[Lid90] W. P. Lidinsky. Data communications needs. *IEEE Network Magazine*, 4(2):28–33, März 1990.

[LKK93] Peter C. Lockemann, Gerhard Krüger und Heiko Krumm. *Telekommunikation und Datenhaltung.* Hanser Studienbücher der Informatik. Carl Hanser Verlag München, 1993.

[LL90] Leslie Lamport und Nancy Lynch. Distributed Computing: Models and Methods. Aus Jan van Leeuwen, Hrsg., *Handbook of Theoretical Computer Science*, Jahrgang B: Formal Models and Semantics, Kapitel 18, Seiten 1157–1199. Elsevier, Amsterdam, 1990.

[LM91] Gilbert M. Lundy und Raymond E. Miller. Specification and analysis of a data transfer protocol using systems of communicating machines. *Distributed Computing*, 5:145–157, 1991.

[LM92] G. M. Lundy und R. C. McArthur. Formal Model of a High Speed Transport Protocol. Aus R. J. Lynn, Jr. und M. Ü. Uyar, Hrsg., *Protocol Specification, Testing, and Verification XII*, Seiten 163–178. Elsevier, 1992.

[LS84] Simon S. Lam und A. Udaya Shankar. Protocol Verification via Projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, April 1984.

[LS91a] Thomas F. La Porta und Mischa Schwartz. Architectures, Features, and Implementation of High-Speed Transport Protocols. *IEEE Network Magazine*, Seiten 14–22, Mai 1991.

[LS91b] Simon S. Lam und A. Udaya Shankar. A Composition Theorem for Layered Systems. Aus B. Jonsson, J. Parrow und B. Pehrson, Hrsg., *Protocol Specification, Testing, and Verification XI*, Seiten 93–108. IFIP, Elsevier, 1991.

[LS91c] K. G. Larsen und A. Skou, Hrsg. *Computer-Aided Verification*, Nummer 575 in Lecture Notes in Computer Science, Berlin Heidelberg, 1991. Springer-Verlag.

[LS92] J. B. Lyles und D. C. Swineheart. The Emerging Gigabit Environment and the Role of Local ATM. *IEEE Communications Magazine*, Seiten 52–58, April 1992.

[LS93] Thomas F. La Porta und Mischa Schwartz. The MultiStream Protocol: A highly flexible high-speed transport protocol. *IEEE Journal on Selected Areas in Communications*, 11(4), Mai 1993.

[LS94] Thomas F. La Porta und Mischa Schwartz. Verification of the MultiStream Protocol (MSP) Using COSPAN. *Formal Methods in System Design*, 4(2):123–153, Februar 1994.

[LT94] G. M. Lundy und H. A. Tipici. Specification and Analysis of the SNR High-Speed Transport Protocol. *IEEE/ACM Transactions on Networking*, 2(5):483–496, Oktober 1994.

[Lüb91] Ralf Lübeck. Interpreter-Unterstützung für den Entwurf verteilter Systeme mit Transition Axioms. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, September 1991.

[MC81] Jayadev Misra und K. Mani Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, Juli 1981.

[McA90] Derek Robert McAuley. Protocol Design for High Speed Networks. Technischer Bericht 186, University of Cambridge, Computer Science Laboratory, Cambridge, UK, Januar 1990.

[McC94] William W. McCune. OTTER 3.0 Reference Manual and Guide. Interner Bericht ANL-94/6, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439-4801, USA, Januar 1994.

[Mei94] Olaf Meier. Ein erweitertes Werkzeug zur rechnergestützten Einarbeitung in TLA-Spezifikationen verteilter Systeme. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, November 1994.

[Mer76] P. M. Merlin. A Methodology for the Design and Implementation of Communication Protocols. *IEEE Transactions on Communications*, 24(6):614–621, Juni 1976.

[Mer79] P. M. Merlin. Specification and Validation of Protocols. *IEEE Transactions on Communications*, 27(11):1671–1680, November 1979.

[Mes93] Arnulf Mester. tTLA$^+$ 1 syntax description. Interner Bericht Dok-TLA92/4, Universität Dortmund, FB Informatik, LS IV, D-44221 Dortmund, Januar 1993.

[MH94] Arnulf Mester und Peter Herrmann. Tools for TLA-based specifications. Interner Bericht Dok-TLA92/35, Universität Dortmund, FB Informatik, LS IV, D-44221 Dortmund, Februar 1994.

[Mil80] Robin Milner. *A Calculus for Communicating Systems.* Nummer 92 in Lecture Notes in Computer Science. Springer, Berlin, 1980.

[Mil83] Robin Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[MK95] Arnulf Mester und Heiko Krumm. Composition and Refinement Mapping based Construction of Distributed Applications. Aus *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Aarhus, Denmark, 1995. BRICS.

[MM91] Wenbo Mao und George J. Milne. An Automated Proof Technique for Finite-State Machine Equivalence. Aus Larsen und Skou [LS91c], Seiten 233–243.

[Mor88] S. P. Morgan. Window Flow Control in a Truncated Byte-Stream Virtual Circuit. *IEEE Transactions on Communications*, 36:816–825, Juli 1988.

[MP87] Zohar Manna und Amir Pnueli. A Hierarchy of Temporal Properties. Technischer Bericht STAN-CS-87-1186, Stanford University, Dept. of Computer Science, Oktober 1987.

[MP92a] Zohar Manna und Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems — Specification.* Springer-Verlag, New York, 1992.

[MP92b] Zohar Manna und Amir Pnueli. Temporal Specification and Verification of Reactive Modules. Interner Bericht, Weizmann Institute, Rehovot, Israel, 1992.

[MS82] J. Martinez und M. Silva. *A Simple and Fast Algorithm to Obtain All Invariants of a Generalized Petri Net*, Seiten 301–310. Nummer 52 in Informatik-Fachberichte. Springer, 1982.

[Muk91] Biswanath Mukherjee. Integrated Voice-Data Communication over High-Speed Fiber Optic Networks. *IEEE Computer*, Seiten 49–58, Februar 1991.

[MV91] S. Mauw und G. J. Veltink. A Proof Assistant for PSF. Aus Larsen und Skou [LS91c], Seiten 158–168.

[NH89] L. H. Ngoh und T. P. Hopkins. Transport Protocol Requirements for Distributed Multimedia Information Systems. *The Computer Journal*, 32(3), 1989.

[NRS90] Arun N. Netravali, W. D. Roome und K. Sabnani. Design and Implementation of a High-Speed Transport Protocol. *IEEE Transactions on Communications*, 38(11):2010–2024, November 1990.

[Och94] Peter Ochsenschläger. Kompositionelle Verifikation kooperierender Systeme. Arbeitspapiere der GMD 885, GMD, Dezember 1994.

[OG76] S. Owicki und D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[OP91] S. W. O'Malley und L. L. Peterson. A Highly-Layered Architecture for High-Speed Networks. Aus M. J. Johnson, Hrsg., *Protocols for High-Speed Networks, II*, Seiten 141–156, IFIP WG 6.1/WG 6.4 Second International Workshop on Protocols for High-Speed Networks, Palo Alto, Ca., 1991. IFIP, North-Holland.

[OP92] S. W. O'Malley und L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, 1992.

[ORS92] S. Owre, J. Rushby und N. Shankar. PVS: A Prototype Verification System. Aus: D. Kapur, Hrsg., *11th International Conference on Automated Deduction (CADE)*, Nummer 607 in Lecture Notes on Artificial Intelligence, Seiten 748–752, 1992.

[Pad83] M. A. Padlipsky. A Perspective on the ARPANET Reference Model. Aus *Proceedings of IEEE Infocom '83*, Seiten 242–253, 1983.

[PGP92] Bjorn Pehrson, Per Gunningberg und Stephen Pink. Distributed Multimedia Applications on Gigabit Networks. *IEEE Network Magazine*, 6:26–35, Januar 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. Aus *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, Seiten 46–57, 1977.

[Pos80] J. Postel. User Datagram Protocol. Interner Bericht RFC 768, Network Information Center, August 1980.

[Pos81] J. Postel. Transmission Control Protocol. Interner Bericht RFC 793, Network Information Center, September 1981.

[Pot95] Hans-Jürgen Potyka. Ein Werkzeug zur Suche von Refinement Mappings zwischen Kommunikationsprotokollen und Diensten. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, Januar 1995.

[PPVW93] T. Plagemann, B. Plattner, M. Vogt und T. Walter. Modules as Building Blocks for Protocol Configuration. Aus *International Conference on Network Protocols, Los Alamitos, California*, IEEE Computer Society Press, 1993.

[Pro89] Protocol Engines, Incorporated. *XTP Protocol Definition Revision 3.4*, 1989.

[Pro93] Projektgruppe 207. Automatische Generierung von Prototypen verteilter Anwendungen — Endbericht. Interner Bericht, Universität Dortmund, D-44221 Dortmund, März 1993.

[Pro94] Projektgruppe 225. Audiovisuelle Animation von verteilten Algorithmen und Kommunikationsprotokollen — Endbericht. Interner Bericht, Universität Dortmund, D-44221 Dortmund, April 1994.

[PT90] Gurudatta M. Parulkar und Jonathan S. Turner. Towards a Framework for High-Speed Communication in a Heterogeneous Networking Environment. *IEEE Network Magazine*, Seiten 19–27, März 1990.

[Raf93] Robert Raffelsiefer. Rechnerunterstützte Erzeugung vereinfachter Sichten von Spezifikationen verteilter Systeme in TLA. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, August 1993.

[RS92] M. N. Ransom und D. R. Spears. Applications of Public Gigabit Networks. *IEEE Network*, 6(2):30–32, 37–40, März 1992.

[Rud82] H. Rudin. An Introduction to Automated Protocol Validation. Interner Bericht RZ 1174, IBM, September 1982. Forschungsbericht.

[Rud86] H. Rudin. Tools for Protocols driven by Formal Specification. Interner Bericht RZ 1525, IBM, Oktober 1986. Forschungsbericht.

[RW82] H. Rudin und C. West. A Validation Technique for Tightly-Coupled Protocols. *IEEE Transactions on Computers*, April 1982.

[Sch91] Axel Schimice. Rechnergestützte Prüfung der Verfeinerungen von Kommunikationssystemen. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, Mai 1991.

[SHGC92] Doug Shepherd, David Hutchinson, Francisco Garcia und Geoff Coulson. Protocol support for distributed multimedia applications. *Computer Communications*, 15(6):359–366, Juli 1992.

[SHP91] John D. Spragins, Joseph L. Hammond und Krzysztof Pawliowski. *Telecommunications — Protocols and Design.* Addison-Wesley, 1991.

[SK93] Miriam Stange und Thomas Kraatz. Ein Werkzeug zur automatischen Verifikation verteilter Systeme per On-the-Fly Model-Checking in TLA. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, September 1993.

[SLU89] Krishnan K. Sabnani, Aleta M. Lapone und M. Ümit Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, 37(9):940–948, September 1989.

[SP90] James P. G. Sterbenz und Gurudatta M. Parulkar. Axon: A High Speed Communication Architecture for Distributed Applications. Aus *Proceedings of IEEE INFOCOM'90*, Seiten 415–425, San Francisco, CA, USA, Juni 1990. IEEE.

[Spi89] J. M. Spivey. *The Z Notation.* International Series in Computer Science. Prentice-Hall, 1989.

[SRC84] J. H. Saltzer, D. P. Reed und D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2:277–288, November 1984.

[SS93] Douglas C. Schmidt und Tatsuya Suda. Transport System Architecture Services for High-Performance Communications Systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–506, Mai 1993.

[SS94] B. K. Schmidt und V. S. Sunderam. Empirical analysis of overheads in cluster environments. *Concurrency: Practice and Experience*, 6(1):1–32, Februar 1994.

[Sta86] Richard Stallman. GNU Emacs Manual. Interner Bericht, Free Software Foundation, 1986.

[Sto93] Christian Stocker. Spezifikation und Validation eines parallelen Transportsubsystems. Diplomarbeit, Universität Karlsruhe (TH), Fakultät für Informatik, Juni 1993.

[Sun83] C. A. Sunshine. Experience with Automated Protocol Verification. Aus *Proceedings of the 3rd International Workshop on Protocol Specification, Testing, and Verification*, Seiten 229–236, 1983.

[Svo89] Liba Svoboda. Implementing OSI Systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, September 1989.

[SW90] Robert M. Sanders und Alfred C. Weaver. The Xpress Transfer Protocol (XTP) — A Tutorial. *Computer Communication Review*, 20(5):67–80, Oktober 1990.

[Tan89] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 2. Ausgabe, 1989.

[Ten89] David L. Tennenhouse. Layered Multiplexing Considered Harmful. Aus H. Rudin und R. Williamson, Hrsg., *Proceedings of IFIP Workshop on Protocols for High-Speed Networks*, Zürich, Schweiz, Mai 1989. IFIP.

[Tsc91] Christian Tschudin. Flexible Protocol Stacks. *Computer Communication Review*, 21(4):197–205, September 1991. ACM SIGCOMM '91 Conference, Zürich.

[Val89] A. Valmari. Stubborn Sets for Reduced State Space Generation. Aus *Proceedings of the 10th International Conference on Application and Theory of Petri Nets*, Bonn, 1989.

[vBo82] Gregor von Bochmann et al.. Experience with Formal Specifications Using an Extended State Transition Model. *IEEE Transactions on Communciations*, 30(12):2506–2513, Dezember 1982.

[vES91] Peter van Eijk und Jeroen Schot. An Exercise in Protocol Synthesis. Interner Bericht TIOS 91/015, University of Twente — Tele-Informatics & Open Systems Group, Mai 1991.

[VL88] S. T. Vuong und T. Lau. VALIRA/VALISYN Protocol Validator/Synthesizer User's Manual (Version 1.2). Interner Bericht 88–19, Department of Computer Science, The University of British Columbia, Vancouver, BC, Kanada, 1988.

[VSvS88] Chris A. Vissers, Guiseppe Scollo und Marten van Sinderen. Architecture and specification style in formal descriptions of distributed systems. Aus S. Agarwal und K. Sabnani, Hrsg., *Protocol Specification, Testing and Verification*, Jahrgang VIII, Seiten 189–204, Elesevier, 1988. IFIP.

[VSvS91] Chris A. Vissers, Guiseppe Scollo und Marten van Sinderen. Specification styles in distributed system design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[vWL92] J. von Wright und T. Långbacka. Using a Theorem Prover for Reasoning about Concurrent Algorithms. Aus v. Bochmann, G. und D. K. Probst, Hrsg., *Computer Aided Verification*, Nummer 663 in Lecture Notes in Computer Science, Seiten 44–55, 4th International Workshop, CAV '92, Montreal, 1992. Springer-Verlag.

[Wea94] Alfred C. Weaver. The Xpress Transfer Protocol. *Computer Communications*, 17(1):46–52, Januar 1994.

[Weg94] Frank Wegmann. Rechnergestütztes Erstellen von Spezifikationen verteilter Systeme in TLA$^+$. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, Juli 1994.

[Wes78] C. West. General Technique for Communications Protocols Validation. *IBM J. Res. Develop.*, 22:393–404, 1978.

[WG84] William M. Waite und Gerhard Goos. *Compiler Construction.* Springer-Verlag New York, 1984.

[WG93] P. Wolper und P. Godefroid. Partial order methods for temporal verification. Aus *4th International Conference on Concurrency Theory*, Nummer 715 in Lecture Notes in Computer Science, Springer-Verlag, 1993.

[Wie94] Holger Wiemann. Ein Werkzeug zur automatischen Verifikation von Safety- und Fairness-Eigenschaften verteilter Systeme. Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, September 1994.

[Win90] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, Seiten 8–24, September 1990.

[Wir85] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag Berlin Heidelberg, 3. Ausgabe, 1985.

[WL93] Pierre Wolper und Denis Leroy. Reliable Hashing without Collision Detection. Aus Costas Courcoubetis, Hrsg., *Computer Aided Verification*, Nummer 697 in Lecture Notes in Computer Science, Seiten 59–70, Elounda, Griechenland, Juni 1993. Springer Verlag Berlin, Heidelberg, New York.

[WM87] Richard W. Watson und Sandy A. Mamrak. Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices. *ACM Transactions on Computer Systems*, 5(2):97–120, Mai 1987.

[Wol82] Pierre Wolper. *Synthesis of Communicating Processes from Temporal Logic Specification*. Doktorarbeit, Stanford University, August 1982.

[Wol89] Pierre Wolper. On the Relation of Programs and Computations to Models of Temporal Logic. Aus H. Barringer B. Banieqbal und A. Pnueli, Hrsg., *Temporal Logic in Specification*, Nummer 398 in Lecture Notes in Computer Science, Seiten 75–123, Altrincham, UK, April 1989. Springer Verlag Berlin, Heidelberg, New York.

[WWM+90] L. Wos, S. Winkler, W. McCune, R. Overbeek, E. Lusk, R. Stevens und R. Butler. Automated Reasoning Contributes to Mathematics and Logic. Aus M. E. Stickel, Hrsg., *Proceedings of the 10th International Conference on Automated Deduction*, Nummer 449 in Lecture Notes in Computer Science, 7 1990.

[XTP95] XTP Forum, 1394 Greenworth Place, Santa Barbara, CA 93105, USA. *XTP Transport Protocol Specification, Revision 4.0*, März 1995.

[YK90] Maria C. Yuang und A. Kershenbaum. Parallel Protocol Verification: The Two–Phase Algorithm. Aus *Protocol Specification, Testing and Verification IX*, Seiten 339–353. North-Holland, 1990.

[Zha86] L. Zhang. Why TCP Timers Don't Work Well. Aus *Proceedings of ACM SIGCOMM'86 Symposium*, Seiten 397–405, Stowe, WT, USA, August 1986. ACM.

[Zit89] Martina Zitterbart. High-Speed Protocol Implementations Based on a Multiprocessor Architecture. Aus H. Rudin und R. Williamson, Hrsg., *Proceedings of IFIP Workshop on Protocols for High-Speed Networks*, Zürich, Schweiz, Mai 1989. IFIP.

[Zit91] Martina Zitterbart. High-Speed Transport Components. *IEEE Network Magazine*, Seiten 54–63, Januar 1991.

[Zit95] Martina Zitterbart. *Hochleistungskommunikation — Band 1: Technologie und Netze*. Oldenbourg Verlag, München, 1995.

[ZST93] Martina Zitterbart, Burkhard Stiller und Ahmed N. Tantawy. A Model for Flexible High-Performance Communication Subsystems. *IEEE Journal on Selected Areas in Communications*, 11(4):507–518, Mai 1993.

[ZWR+80] P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan und D. Brand. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, 28(4):651–661, April 1980.

Tabellenverzeichnis

2.1 Protokollfunktionen und Fehlerbehebungsmaßnahmen in MSP 27

5.1 Theoreme zum Beweis der Safety-SCs des Datentransfers 144

6.1 Bedeutung der Datentransfermodus-Parameter in FAPMs 164
6.2 Komposition der Datentransfer-FAPMs 172
6.3 Dienste der Intraconnection-Coordination 173
6.4 Komposition der FAPMs der Signalisierungsdatenkanäle 181
6.5 Komposition der Datentransfer-SC 190
6.6 Komposition der SCs der Signalisierungsdatenkanäle 194

Abbildungsverzeichnis

1.1 Aufbau eines Frameworks zur formal basierten Systementwicklung 3

2.1 Kommunikationsdienst und -protokoll 8
2.2 Transportsystem-Architektur 14
2.3 Methoden zur Beschleunigung von Kommunikationssystemen 15
2.4 HOPS-Struktur und TCP/IP-Protokollgraph 20
2.5 Evolution der Hochleistungstransportprotokolle 23

3.1 TLA 44
3.2 Axiome und Beweisregeln von TLA 45
3.3 Quantoren in TLA 46
3.4 TLA$^+$-Spezifikationsbeispiel 58
3.5 Temporale TLA$^+$-Operatoren 60
3.6 Konstante TLA$^+$-Operatoren 61
3.7 TLA$^+$-Spezifikation eines Verfeinerungsbeweis 62

4.1 Systemkomposition durch gekoppelte Aktionen 66
4.2 Strukturierte Verifikation kompositionaler Spezifikationen 76
4.3 Umformungen auf cTLA$^+$-Prozessen 82
4.4 Szenario eines Fairnessbeweises 94

5.1 Aufbau des Frameworks 97
5.2 Komposition von FAPMs und AMCs zu Protokollspezifikationen 99
5.3 Aufbau der Datentransferkanäle 101

6.1 XTP-Verbindungsabbauverfahren für Duplex-Datenkanäle 158
6.2 XTP-PDU-Format 160
6.3 Ablaufverhalten der XTP-Dienstelemente 174
6.4 Verifikation der SCs für FAPM *SSynchronize* 191