TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Eine Methode zur formalen Modellierung von Betriebssystemkonzepten

Katharina Spies

TUM-I9809
SFB-Bericht Nr. 342/03/98 A
Juni 98

TUM–INFO–06-I9809-150/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Fakultät für Informatik
der Technischen Universität München

Eine Methode zur formalen Modellierung von Betriebssystemkonzepten

Katharina Spies

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. E. Jessen
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. M. Broy
	2.	Univ.-Prof. Dr. U. Baumgarten

Die Dissertation wurde am 17.12.1997 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 9.2.1998 angenommen.

Kurzfassung

FOCUS umfaßt allgemeine Konzepte, Beschreibungs- und Verfeinerungstechniken zur Top-Down-Entwicklung verteilter, reaktiver Systeme. Mit seiner präzisen, formalen Basis ist FOCUS überall dort einsetzbar, wo ein System als Netz von interagierenden Komponenten modelliert werden kann.

Betriebssysteme sind langlebige Softwareprodukte, die von größter Wichtigkeit und für die Nutzung von Rechensystemen unverzichtbar sind. Sie sind Systeme mit einer großen Anzahl interagierender Komponenten und bieten mit den Problemen der Ressourcenverwaltung sowie in bezug auf Strukturierung und Abstraktion eine Vielzahl interessanter Aufgabenstellungen. Die systematische Entwicklung von Betriebssystemen mit formalen Verfahren wurde bisher nur wenig betrachtet und ist für den Einsatz von FOCUS eine Herausforderung.

Die vorliegende Arbeit zeigt, daß das semantische Modell von FOCUS dazu geeignet ist, Betriebssysteme zu modellieren. Es werden Spezifikationsmuster zur schematischen Erstellung formaler Spezifikationen definiert. Die Spezifikationserstellung orientiert sich zum einen an der systematischen und einheitlichen Darstellung der Formalisierungen. Zum anderen findet das Prinzip Verwendung, einfache Systeme mit Kernfunktionalität zu modellieren und zu diesem schrittweise weitere Funktionalitäten hinzuzunehmen, wobei bereits erstellte Spezifikationen konsequent erweitert, angepaßt und weiterentwickelt werden.

Die methodische Vorgehensweise zur Entwicklung einer Systemspezifikation basiert auf einem Kernsystem zur Modellierung von Konzepten der Prozessorverwaltung. Das Kernsystem wird schrittweise erweitert um Konzepte der Speicherverwaltung mit virtuellem Speicher, Prozeßkooperation durch Nachrichtenaustausch sowie die Erzeugung von Prozessen und deren Integration in das bestehende System. Als Ergebnis erhalten wir ein in FOCUS modelliertes *Betriebssystem*, mit dem wesentlichen Teile der Ressourcenverwaltung auf hohem Abstraktionsniveau beschrieben sind. Durch den konsequenten Einsatz der Spezifikationsmuster entstehen Spezifikationen in einem einheitlichen Format. Aufgrund der systematischen Vorgehensweise kann die vollständige Modellierung in kleinen an speziellen Betriebssystemaufgaben orientierten Schritten nachvollzogen werden.

Danksagung

Bei Prof. Dr. Manfred Broy bedanke ich mich herzlich für seine Anleitung und Betreuung während der letzten Jahre und vor allem für die Möglichkeit, mich mit der Modellierung von Betriebssystemen befassen zu können. Ich danke ihm und Prof. Dr. Uwe Baumgarten für das Durchsehen von Vorversionen der Arbeit und ihre Kommentare hierzu.

Besonders ausdrücklich und herzlich bedanke ich mich bei Ursula Hinkel, die immer bereit war, mit mir während dem Entstehen dieser Arbeit zu diskutieren und die Vorversionen sorgsam durchzusehen. Ebenso gilt mein besonderer Dank Bernhard Schätz und Prof. Dr. P.P. Spies für intensive Diskussionen und die Beantwortung vieler Fragen zu Focus bzw. Betriebssystemen. Alle drei haben wesentlich zum Gelingen dieser Arbeit beigetragen.

Herzlich bedanke ich mich bei Max Breitling für das aufmerksame und zügige Durchsehen einer Vorversion der gesamten Arbeit. Für das sorgsame Durchlesen einer Vorversion des Focus-Kapitels und hilfreiche Kommentare danke ich Franz Huber, Volkmar Lotz, Monika Schmidt und Veronika Thurner. Dank gilt auch Franz Huber, Bernhard Schätz, Alexander Schmidt und Oscar Slotosch für das Durchsehen einzelner Kapitel.

Nicht zuletzt danke ich meinen Eltern Karin und Peter Paul Spies für ihre liebevolle moralische Unterstützung vor allem während des vergangenen Jahres.

Inhaltsverzeichnis

Kapitel 1

Einführung

Systeme zur Steuerung industrieller Produktionen oder Betriebssysteme als Basis für die Nutzung einer Rechenanlage sind klassische Anwendungsgebiete der praktischen und systemnahen Informatik. Oftmals entsprechen sie verteilten Systemen, deren Funktionalität durch das Zusammenspiel einer Vielzahl von Komponenten, die verschiedenste Teilaufgaben bearbeiten, gewährleistet wird. Die Entwicklung von Software für derartige Systeme erfordert einen systematischen und strukturierten Konstruktionsprozeß, um der Komplexität der Aufgabenstellung gerecht zu werden. Die Beschreibung eines Systems auf hohem Abstraktionsniveau gibt die Möglichkeit, die Anforderungen an das System festzulegen, ohne sich direkt mit realisierungstechnischen Details befassen zu müssen.

Formale Methoden und Techniken bieten aufgrund ihrer mathematisch-logischen Fundierung die Grundlage zur Entwicklung verteilter Systeme auf hohem Abstraktionsniveau. Sie stellen formal abgesicherte Techniken bereit, die eine präzise Spezifikation und den formalen Nachweis der Gültigkeit von Systemeigenschaften erlauben. Zur Behandlung mehrerer Entwicklungsstufen können Spezifikationen mit zunehmendem Detaillierungsgrad zueinander in formal abgesicherte Relationen, sogenannten Verfeinerungsbeziehungen, gestellt werden. Formale Methoden sind vor allem darauf ausgerichtet, Systeme zu entwickeln, an die hohe Sicherheits- und Qualitätsanforderungen gestellt werden.

Bisher verfügen solche Methoden noch nicht über die nötige Reife, um sie zur vollständigen Entwicklung industrieller Softwareprodukte für technische Systeme einzusetzen. Dies liegt nicht zuletzt daran, daß im Rahmen formaler Methoden meist komplexe und keineswegs leicht verständliche Konzepte verwendet werden, deren Einsatz eine gewisse Übung im Umgang mit mathematischen Formulierungen erfordert. Damit können sie von Anwendern, die mit den Techniken nicht vertraut sind, nicht oder nur mit großem Aufwand eingesetzt und verwendet werden. Im allgemeinen wirken sich mathematisch-logische Techniken und Denkweisen jedoch positiv auf die Strukturierung und Klarheit einer Realisierung aus.

Bereits das Erstellen einer ersten Formalisierung stellt meist eine schwer zu bewältigende Hürde dar. Dieser Schritt ist mit großer Sorgfalt durchzuführen, da eine Systementwicklung auf dieser Formalisierung, die die wesentlichen Anforderungen an das System auf abstrakte Weise umfaßt, basiert. Diese Anforderungen müssen erarbeitet und festgelegt werden, da Systembeschreibungen üblicherweise nicht in geeigneter Form vorliegen. Diese Erarbeitung sollte von Kennern einer formalen Methode gemeinsam mit Kennern der zu bearbeitenden Aufgabe durchgeführt werden. Da Anwender meist nur über geringe Kenntnisse im Umgang mit formalen Methoden verfügen, ist hier von der Seite der Entwickler formaler Methoden Hilfe bereitzustellen, um das Erlernen der speziellen Notationen zu erleichtern.

Die Anwendbarkeit formaler Methoden wurde bisher von deren Entwicklern vor allem an Beispielen gezeigt, die aus theoretischer Sicht interessant sind und die Mächtigkeit der Methoden demonstrieren. Oftmals haben diese Beispiele jedoch aus Anwendersicht zu wenig Bezug zu *realen* Problemen und deren Vielschichtigkeit. Da offensichtlich ist, daß sowohl die Anwender als auch die theoretischen Arbeiten von einer Verflechtung profitieren können, müssen zukünftige Arbeiten darauf abzielen, hier einen Brückenschlag vorzunehmen.

1.1 Zielsetzung

Zu Beginn dieses Kapitels wurde erklärt, daß es zur Weiterentwicklung ausgereifter formaler Methoden und Techniken wesentlich ist, deren Praxistauglichkeit anhand großer Anwendungen zu überprüfen und nachzuweisen. Daraus und mit dem Vorsatz, eine derartige Spezifikationsentwicklung systematisch und unter methodischen Gesichtspunkten durchzuführen, erklärt sich die Zielsetzung der vorliegenden Arbeit.

> Unter Verwendung des formalen Rahmens von Focus ist ein verteiltes Systems zu spezifizieren, dessen Verhalten der grundlegenden Funktionsweise eines Betriebssystems auf hohem Abstraktionsniveau entspricht. Die Entwicklung der formalen Modellierung soll schrittweise und systematisch erfolgen.

Focus, siehe auch [BDD+92], [BDD+93] und [BS98], ist ein am Lehrstuhl Broy erarbeiteter Rahmen zur Entwicklung verteilter, reaktiver Systeme. Fundiert durch eine präzise und formale Basis ist Focus überall dort einsetzbar, wo ein System als Netz von interagierenden Komponenten modelliert werden kann. Die Komponenten kommunizieren asynchron über gerichtete Kanäle und werden durch Mengen stromverarbeitender Funktionen modelliert. Gemäß Focus geht eine Systementwicklung von der Erarbeitung der Anforderungen und deren Umsetzung in eine erste Focus-Spezifikation aus und führt über mehrere Entwicklungsstufen mit wachsendem Detaillierungsgrad hin zu einem lauffähigen Programm. Zur Formalisierung steht eine breite Palette von Spezifikationsformaten, siehe beispielsweise [Bro94], [FS93] und [BS98], und (anwendungsorientierten) Beschreibungstechniken, siehe [HSS96], [SS95] oder [Spi94], zur Verfügung. Verfeinerungsbegriffe, siehe [Bro92b] und [BS98], erlauben die Hinzunahme von Details, die sich auf verschiedene Aspekte (Struktur, Daten, Verbindungskanäle) des modellierten Systems beziehen. Einzelne Komponenten können separat spezifiziert und entwickelt werden, um sie dann zum gewünschten System zusammenzusetzen. Focus läßt Zustandsmodellierungen zu und bietet die Möglichkeit zur Modellierung mobiler und dynamischer Systemstrukturen, siehe [GS97] und [HS96]

Betriebssysteme sind langlebige Softwareprodukte, die von größter Wichtigkeit und für die Durchführung von Berechnungen mit Rechensystemen unverzichtbar sind. Sie sind Systeme mit einer großen Anzahl interagierender Komponenten und gehören zu den Informatikanwendungen, mit der die meisten Benutzer eines Rechensystems bereits in Berührung gekommen sind. Betriebssysteme bieten mit ihren Algorithmen und der Ressourcenverwaltung, aber auch bzgl. Strukturierung und Abstraktion eine Vielzahl interessanter Aufgabenstellungen. Da die Entwicklung von Betriebssystemen zu den klassischen Aufgaben der Informatik gehört, wurden sie bereits früh und entsprechend dem damaligen Stand der Softwareentwicklung realisiert. Viele der verbreiteten Betriebssysteme wurden nach Bedarf Stück für Stück verändert und verbessert. Gerade wegen ihrer Wichtigkeit wird es für die Entwicklung zukünftiger Betriebssysteme unerläßlich sein, formale Methoden einzusetzen, um dem hohen Qualitätsanspruch gerecht zu werden. Dies gilt vor allem für die Bereiche, in denen spezifische, auf bestimmte Anwendungsbereiche zugeschnittene Betriebssysteme entwickelt werden, wie beispielsweise in Steuergeräten von Kraftfahrzeugen. Aus diesen Gründen und mit der zu Anfang des Abschnitts ausgeführten Motivation haben wir uns für die formale Modellierung von Betriebssystemkonzepten als geeignetem Kandidat für einen Brückenschlag zwischen Focus und großen Anwendungen entschieden.

Die Zielsetzung, ein Betriebssystem als verteiltes Focus-System zu modellieren, erfordert die Erarbeitung einer allgemeinen Charakterisierung von Betriebssystemen und der zu modellierenden Betriebssystemkonzepte auf hohem Abstraktionsniveau, wofür die Betriebssystemliteratur wie [PS85] und [GS94], [Tan90] und [Tan94] die Basis bildet. Dabei sind zwei Seiten zu berücksichtigen: Zum einem soll die Charakterisierung als allgemeine Abstraktion der technischen Realisierungen von Betriebssystemen gelten können. Zum anderen muß sie jedoch der gemäß Focus benötigten Sichtweise auf verteilte Systeme entsprechen. Damit kann die vorliegende Arbeit aufgrund der erarbeiteten Darstellung auch allgemein zu einem besseren Verständnis der speziellen Betriebssystemaufgaben und -konzepte beitragen. Dies gilt vor allem deshalb, weil eine Beschreibung zu erarbeiten ist, in der auf technische oder realisierungsabhängige Details verzichtet wird.

Die Erstellung der formalen Modellierung soll unter methodischen Gesichtspunkten erfolgen, um allgemeine Hinweise zu erhalten, wie Focus zur Entwicklung großer Anwendungen eingesetzt werden kann. Mit der hier gewählten Vorgehensweise sollen die Grundlagen, die verwendeten Beschreibungstechniken und die für Focus charakteristische Sicht auf verteilte Systeme erklärt werden. Die Behandlung einer konkreten Anwendung soll es ermöglichen, daß ein mit Focus wenig vertrauter Leser Aufgabenstellungen aus dem Bereich der Betriebssysteme oder verwandten Gebieten, die eine ähnliche Modellierung erlauben, eigenständig bearbeiten und mit den Focus-Techniken modellieren kann.

1.2 Ergebnisse

Der allgemeine Rahmen von Focus und die semantische Basis wurden mit dieser Arbeit einer Prüfung unterzogen. Mit der Bearbeitung einer derartig komplexen Anwendung wurde gezeigt, daß Focus zur systematischen Spezifikationsentwicklung für komplexe Software erfolgreich eingesetzt werden kann. Die Spezifikationen, die im mathematisch-logischen Stil erstellt sind, wurden so aufbereitet, daß sie für Leser, die mit Focus nicht vertraut sind, nachvollziehbar und verständlich sind. Die Betriebssystemkonzepte wurden auf hohem Abstraktionsniveau und ohne Einbeziehung von Realisierungsdetails behandelt, sind jedoch aus Betriebssystemsicht durchaus realistisch, was die Erarbeitung der Anforderungen der einzelnen Teilaufgaben anhand der zitierten Literatur zeigt. Die Systematik der Spezifikationserstellung orientiert sich sowohl an der systematischen und einheitlichen Darstellung der Formalisierungen als auch an dem Prinzip, einfache Systeme mit Kernfunktionalität zu modellieren und diese Schritt für Schritt um weitere Funktionalitäten zu erweitern, wobei bereits erstellte Spezifikationen konsequent erweitert und angepaßt werden. Im einzelnen wurden folgende Ergebnisse erzielt:

Die semantischen Konzepte und Beschreibungstechniken von Focus, die für das Verständnis der erstellten formalen Modellierungen notwendig sind, wurden auf eine Art und Weise dargestellt und beschrieben, mit der

1. einem Entwickler Unterstützung zur Erarbeitung formaler Modellierungen gegeben wird. Durch die hier neu entwickelten und festgelegten Spezifikationsschemata können Spezifikationen mittels rekursiver Funktionsgleichungen ausgehend von einem strukturierten Text systematisch erstellt werden. Die Schemata werden bei der Erstellung der später gezeigten Formalisierungen konsequent eingesetzt.

2. die Grundlagen anschaulich erläutert sind, um einem mit Focus wenig vertrauten Leser den Zugang zu den Formeln zu erleichtern und diese verständlich zu machen. Die semantische Basis ausgehend von den Strömen über das gezeitete Modell, den statischen Systemen bis zur Modellierung mobiler, dynamischer Strukturen, ist in dieser Form erstmals zusammenfassend dargestellt.

Die wesentlichen Charakteristika, Aufgaben und Bestandteile eines Betriebssystems werden allgemein, aber auch auf die entwickelte formale Modellierung zugeschnitten, erläutert. Diese abstrakte Beschreibung, die mit der Spezifikationserstellung Hand in Hand geht, liefert eine einfache und gut verständliche Darstellung von zentralen Betriebssystemkonzepten bzgl. der in einem Rechensystem auftretenden Managementaufgaben.

Insgesamt zeigt sich an den Modellierungen die Stärke der gewählten Vorgehensweise:

> Die Weiterentwicklung der Formalisierung orientiert sich an der steigenden Komplexität der Aufgabenstellung. Aufgrund vorbereitender Maßnahmen in „einfachen“ Spezifikationen entstehen deren Erweiterungen durch Anpassungen und Vervollständigungen. Die formalen Modellierungen der Systeme mit wachsender Komplexität

sind schrittweise nachvollziehbar und erscheinen dadurch leichter verständlich, als die direkte Modellierung des Systems in seinem vollen Umfang.

Die Erstellung von Spezifikationen für Systeme mit komplexem Verhalten bleibt prinzipiell schwierig. Es ist jedoch von Vorteil, sich den vollständigen Spezifikationen ausgehend von einem in seiner Funktionalität stark reduzierten System schrittweise zu nähern. Anhand einer, bezogen auf den Anwendungsbereich, vergleichsweise einfachen Aufgabenstellung wird eine erste Spezifikation erarbeitet. Das weitere Vorgehen ist dadurch gekennzeichnet, daß die Funktionalität erweitert und dies auf die formalen Spezifikationen übertragen wird. Die Erstellung der Formalisierungen ist nachgeordnet, und der Entwickler kann sich auf den Inhalt der ihm gestellten Aufgabe konzentrieren. Bestehende Spezifikationen können Schritt für Schritt nachvollzogen werden, und die Veränderungen an den Formalisierungen entsprechen den aufgrund der erweiterten Funktionalität zu erwartenden Anpassungen ohne zusätzliche durch Focus begründete Maßnahmen zu ergreifen. Die insgesamt beschrittene Vorgehensweise ist auch aus der Sicht der Betriebssysteme beachtenswert, da sich viele für Betriebssysteme spezifische Konzepte mit den für die Formalisierung in Focus herausgearbeiteten Informationen gut erklären lassen.

1.3 Methodische Vorgehensweise

Die methodische Entwicklung der Modellierung eines Systems zur Betriebsmittelverwaltung mit den Bereichen Prozessor- und Speicherverwaltung, Prozeßkooperation und -verwaltung wurde in folgenden Schritten durchgeführt:

Das Kernstück der Arbeit bildet die Modellierung von Konzepten der **Prozessorverwaltung**. Insgesamt ergibt sich die vollständige Modellierung der Betriebssystemkonzepte gemäß der Stück für Stück erweiterten Aufgabenstellung durch Ergänzungen, Anpassungen und Weiterentwicklungen dieser ersten Formalisierungen. Wir betrachten zunächst ein System, in dem Prozesse ausschließlich das Betriebsmittel *Prozessor* benötigen, um die Ausführung ihrer Berechnung voranschreiten zu lassen. Die Entscheidung für den so gewählten Ausgangspunkt und der daraus resultierende strukturelle Aufbau des Systems sind durch folgende Punkte charakterisiert:

1. Prozesse sind das zentrale Konzept zur Beschreibung eines Programms in Ausführung. Die Aufgaben der Betriebsmittelverwaltung werden anhand der Zustandsübergänge und damit des Verhaltens eines Prozesses verdeutlicht.

2. Ein Prozessor ist das wesentliche Betriebsmittel, ohne das keine Berechnung von einem Rechensystem ausgeführt werden kann. Da der Prozessor der *Motor* eines Rechensystems ist, stellt eine Komponente zur Modellierung eines Prozessors ebenfalls den Motor des modellierten Systems dar.

Prozessoren und Prozesse werden jeweils als Komponenten modelliert. Dabei wird die Analogie zwischen den Zustandsübergängen der Prozesse und den für die Ressourcenverwaltung charakteristischen Schritten bei der Ausführung einer Berechnung durch den Prozessor – atomare Ausführung der Instruktionen, Adressenbestimmung, Unterbrechung bei fehlenden Ressourcen – explizit am Verhalten kooperierender Komponenten aufgezeigt.

Der zentralen Anteil der Prozessorverwaltung umfaßt die Auswahl des als nächstes auszuführenden Prozesses mittels des bekannten Schedulingverfahrens *Round-Robin*. Ein so ausgewählter Prozeß wird an den Prozessor gebunden, diese Bindung nach einer festgelegten Zeitspanne gelöst und der Prozessor einem nächsten Prozeß zugeteilt. Diese Aufgaben übernimmt ein Dispatcher. Gemäß unserer allgemeinen Zielsetzung, vollständige Spezifikationen ausgehend von einfachem bis hin zu komplexem Verhalten methodisch zu entwickeln, umfaßt die Modellierung der Prozessorverwaltung folgende Entwicklungsschritte:

1. Die Modellierung eines Systems, das der einfachst möglichen Konkurrenzsituation entspricht: Es besteht aus einem Prozeß, einem Prozessor und vorbereitend aus allen für die Prozessorverwaltung erforderlichen Verwaltungskomponenten.

2. Die Funktionalität wird schrittweise so erweitert, daß immer eine feste Anzahl und zunächst zwei und dann viele Prozesse um einen Prozessor konkurrieren.

3. Das System mit einer festen Anzahl von Prozessen wird um die Verwaltung von mehreren Prozessoren erweitert. Hierbei werden alternativ sowohl deren zentrale als auch deren dezentrale Verwaltung spezifiziert.

Die erste Erweiterung der Funktionalität des Systems aus der Sicht der Aufgaben des Ressourcenmanagements bezieht sich auf die **Speicherverwaltung**. Das geforderte Verhalten und die Systemstruktur sind dadurch charakterisiert, daß ein Prozeß nun zusätzlich zum Prozessor auch das Betriebsmittel *Speicherplatz* benötigt, um die ihm zugeordnete Berechnung ausführen zu können.

Die Konzepte eines virtuellen Speichers mit einem prozeßlokalen Seitenersetzungsverfahren gemäß einer LRU-Strategie sowie die Berechnung physikalischer Adressen werden hinzugenommen. Es werden folgende Aufgaben gelöst:

1. Den einzelnen Berechnungsschritten werden virtuelle Adressen zugeordnet.

2. Virtuelle Adressen werden interpretiert, und ein Prozeß wird blockiert, wenn die zugehörige Seite nicht im Arbeitsspeicher eingelagert ist.

3. Die Systemspezifikation wird um neue Komponenten erweitert, die für die neu zu bewältigenden Verwaltungsaufgaben, wie beispielsweise die Berechnung physikalischer Adressen, zuständig sind.

4. Die bestehenden Modellierungen werden angepaßt sowie die Komponenten, für deren Spezifikationen keine Anpassungen nötig sind, übernommen.

In der Modellierung soll die Spezifikation einer großen Anzahl *zentraler* Komponenten, die von allen Informationen erhalten und diese für alle verwalten müssen, vermieden werden. Die von uns gewählte Vorgehensweise ist vor allem dadurch charakterisiert, möglichst viele

Informationen, die ausschließlich lokal benötigt werden, durch entsprechend lokal zugeordnete Verwaltungskomponenten zu modellieren. Daraus resultiert auch die Entscheidung für die Spezifikation eines prozeßlokalen Verfahrens.

In einem nächsten Entwicklungsschritt werden Konzepte der **Prozeßkooperation** mittels Nachrichtenaustausch und den Alternativen *blockierendes Senden* und *nichtblockierendes Senden* zum Verhalten hinzugenommen. Die im System existierenden Prozesse werden zu Gruppen zusammengeschlossen, deren Mitglieder miteinander kooperieren können. Jeder derartigen Gruppe ist eine Komponente zugeordnet, die für die ordnungsgemäße Abwicklung der Kommunikation sorgt. Die Erstellung der beiden alternativen Modellierungen wird durch die methodische Vorgehensweise erleichtert und auch hier zeigt sich, daß nur solche Anteile der Formalisierung, die von der Änderung des Verhaltens direkt betroffen sind, angepaßt bzw. verändert werden müssen.

Abschließend wird das Verhalten durch die **Prozeßerzeugung** vervollständigt. Basierend auf den durch die Umgebung an das System gesendeten Beschreibungen der auszuführenden Berechnungen werden Prozesse erzeugt und in das bestehende System integriert. Es wird überprüft, ob für deren Erzeugung jeweils genügend Speicherplatz zur Verfügung steht, und falls dies der Fall ist, wird dieser belegt. Einem neuen Prozeß wird die auszuführende Berechnung zugewiesen, und er wird der Gruppe zugeordnet, mit deren Mitgliedern er kooperieren kann. Zusätzlich wird gewährleistet, daß nur solche Berechnungen vom System ausgeführt werden, die von Benutzern mit Zugangsberechtigung stammen. Zu diesem Zweck erhält das System einen Pförtner.

1.4 Vergleich mit anderen Arbeiten

Die in der Motivation zu dieser Arbeit beschriebene Kluft zwischen formalen Methoden und der Notwendigkeit, den Nachweis der Praxistauglichkeit durch die Behandlung großer Anwendungen zu überprüfen, ist in dieser Form erst in jüngerer Zeit in den Vordergrund getreten. Die formale Behandlung von Betriebssystemen mit dem vollen Umfang ihrer geforderten, charakteristischen Funktionalität hat dabei als Beispiel für ein komplexes, verteiltes System eine nur unwesentliche Rolle gespielt. Die Notwendigkeit, Betriebssysteme mit formalen Mitteln zu behandeln, um spezifische Produkte von hoher Qualität für bestimmte Anwendungsbereiche und in standardisierter Form zu entwickeln, wurde bisher wenig beachtet. Es gibt nur wenige Arbeiten, die sich mit diesem Thema beschäftigen. Als vergleichende Arbeiten sind daher Arbeiten zu formalen Methoden und den existierenden Ansätzen, die Betriebssystemliteratur und einige Arbeiten, die sich mit speziellen Ausschnitten eines Betriebssystems und deren formaler Modellierung beschäftigen, zu nennen.

In den vergangenen Jahren wurden weltweit zahlreiche formale Methoden und Techniken zur Beschreibung und Entwicklung verteilter Systeme entwickelt, die vergleichbar mit Focus über eine fundierte semantische Basis verfügen. Als Beispiele seien hier TLA ([Lam94]), UNITY ([CM88]) und ProCoS ([BHL+96]) genannt. Für einen Vergleich dieser und weiterer formaler Methoden (wie beispielsweise temporallogische und algebraische Ansätze, Petrinetze, I/O-Automaten oder CSP-artige Ansätze) anhand einer Fallstudie wird auf [BMS96a] und dabei insbesondere auf den vergleichenden Artikel [BMS96b] verwiesen. Die wichtigste Literatur zu FOCUS wird in Kapitel 2 angegeben.

Zu Betriebssystemen und die sie charakterisierenden Konzepte und Verfahren liegt eine Fülle von Literatur vor. Wir nennen hier nur die oft zitierten Standardwerke „Moderne Betriebssysteme“ von A.S. Tanenbaum, [Tan92] und [Tan94], und „Operating Systems Concepts“ von A. Silberschatz, [PS85] und [GS94]. Die Aufgaben eines Betriebssystems werden hier jedoch weder abstrakt noch mittels formaler Techniken beschrieben, sondern liegen in textueller und oftmals stark an Implementierungen und speziellen Algorithmen orientierter Form vor. Wir haben die Anforderungen an die von uns modellierten Konzepte aus den diversen Standardwerken abgeleitet und gehen in Kapitel 3 detaillierter auf diese Literatur ein. Die in den Titeln einiger dieser Arbeiten auftretende Bezeichnung „Theorie“, wie beispielsweise in [CD73], bezieht sich nicht auf den Einsatz formaler Methoden zur Entwicklung der Software, sondern beispielsweise auf die Bewertung von Schedulingstrategien auf der Basis wahrscheinlichkeitstheoretischer Ansätze oder auf die Untersuchung der Effizienz von Algorithmen. Eine Beschreibung von Betriebssystemen mit einem modellorientierten und spurbasierten Ansatz liegt mit [Spi98] vor.

Zur Modellierung einiger Aspekte von Betriebssystemen mit FOCUS wurden mit [BD91] und [Den95] erste Arbeiten geleistet. Im Rahmen der Definition eines Verfeinerungsbegriffs für zustandsorientierte Spezifikationen wurden die Modellierung der Prozeßerzeugung und eines Prozessors beispielhaft bearbeitet. Insgesamt wird in [Den95] jedoch vor allem der Übergang von funktionalen Spezifikationen mit impliziten Zuständen zu Systembeschreibungen mit expliziten Zuständen behandelt. Ein abstrakter *Remote Procedure Call* wird in [Stø96] bzw. [Bro96] im relationalen bzw. funktionalen FOCUS-Stil spezifiziert. In [Bro95b] wurde eine funktionale Spezifikation der Alpha AXP Architektur und spezieller Eigenschaften des Modells für gemeinsamen Speicher modelliert. Als Beispiele für die Behandlung von Betriebssystemaspekten mittels funktionaler Techniken und funktionaler Programmierung seien [JS89] sowie [Sto86] und [Sta91] mit einem auf UNITY basierenden Ansatz genannt. Hierbei stehen jedoch spezielle Betriebssystemaspekte und nicht, wie in der vorliegenden Arbeit, die systematische Spezifikationsentwicklung eines mit formalen Techniken vollständig modellierten *Betriebssystems* im Vordergrund.

1.5 Aufbau der Arbeit

Aus der Aufgabenstellung und der in Abschnitt 1.1 dargestellten methodischen Vorgehensweise ergibt sich der Aufbau der Arbeit, den wir im folgenden angeben.

In **Kapitel 2** werden die semantischen Konzepte und formalen Beschreibungstechniken von FOCUS dargestellt und beschrieben, die für das Verständnis der später erstellten Formalisierungen notwendig sind. Die in der Arbeit eingesetzten Spezifikationsschemata werden eingeführt und festgelegt. Das Kapitel ist in erster Linie für Leser geschrieben, die mit FOCUS nicht vertraut sind.

Kapitel 3 enthält eine allgemeine, aber bereits auf die anschließend entwickelte formale Modellierung zugeschnittene Beschreibung der Aufgaben eines Betriebssystems.

Das Kernstück der Arbeit bildet die formale Modellierung von Konzepten der Prozessorverwaltung in **Kapitel 4**. In Abschnitt 4.3 wird ein System modelliert, das aus einem Prozeß, einem Prozessor und den benötigten Verwaltungskomponenten besteht. In weiteren Schritten spezifizieren wir Systeme, in denen zunächst zwei (Abschnitt 4.4) und dann n Prozesse (Abschnitt 4.5) um einen Prozessor konkurrieren. Abschließend wird ein System mit mehreren Prozessoren behandelt, wobei alternativ sowohl deren zentrale (Abschnitt 4.6) als auch deren dezentrale Verwaltung (Abschnitt 4.7) formal modelliert werden.

Kapitel 5 beinhaltet die Erweiterung des Verhaltens um Aufgaben der Speicherverwaltung basierend auf dem in Kapitel 4 eingeführten System mit einer festen Anzahl von Prozessen und einem Prozessor. Wir modellieren Konzepte eines virtuellen Speichers mit einem prozeßlokalen Seitenersetzungsverfahren gemäß einer LRU-Strategie.

Mit **Kapitel 6** wird das Verhalten um Konzepte der Prozeßkooperation ergänzt. Wir modellieren alternativ die möglichen Kommunikationsformen zum Nachrichtenaustausch: blockierendes Senden mit Rendezvous und nichtblockierendes Senden.

In **Kapitel 7** wird das System um die Prozeßerzeugung erweitert. Zur Ausführung der von Benutzeraufträgen werden Prozesse erzeugt und in das bestehende System integriert. Um zu gewährleisten, daß Aufträge von Benutzern mit Zugangsberechtigung bearbeitet werden, wird das System um eine Komponente mit Pförtnerfunktion erweitert.

Im **Kapitel 8** werden die erzielten Ergebnisse kritisch bewertet und zusammengefaßt; zudem wird ein Ausblick auf mögliche anschließende Arbeiten gegeben.

Kapitel 2

Formale Grundlagen und Schemata

Der Einsatz formaler Methoden ermöglicht die Entwicklung von Systemen, deren Verhalten unter Verwendung mathematisch-logischer Mittel präzise beschrieben wird. Die ersten Schritte einer Systementwicklung bestehen darin, ausgehend von einer (informell) vorgegebenen und möglicherweise unvollständigen Beschreibung der Aufgabenstellung mittels ausgewählter formaler Techniken eine Spezifikation zu erstellen, die die Aufgabe präzisiert. Eine erste Formalisierung muß mit großer Sorgfalt entwickelt werden, da eine vollständige Systementwicklung auf diesen ersten Spezifikationen basiert. Die Erfahrung im Umgang mit formalen Methoden lehrt, daß dieser erste Schritt bereits eine oftmals schwer zu bewältigende Hürde darstellt. Die vorliegende Arbeit wird anhand einer konkreten Anwendung Hilfestellung zur systematischen Spezifikationsentwicklung mit Focus geben.

Focus verfügt über eine fundierte semantische Basis mit verschiedenen Ausprägungen und bietet eine breite Palette mathematisch-logischer, tabellarischer, graphischer und programmiersprachlicher Beschreibungstechniken. Die graphischen Techniken wurden für statische Systemstrukturen definiert und können bisher nur teilweise zur Spezifikation variabler Strukturen eingesetzt werden. Anhand der von uns durchgeführten Anwendung wird zudem deutlich, daß graphische Darstellungen für Systeme mit einer großen Anzahl von Komponenten unübersichtlich werden können. Wir verwenden graphische Techniken, soweit sie zur Veranschaulichung beitragen, erstellen die Spezifikationen jedoch mittels Funktionsgleichungen. Hierfür werden Schemata definiert, die unabhängig von der Modellierung von Betriebssystemkonzepten einsetzbar sind. Die Beschreibung der Aufgabenstellung wird dabei in eine strukturierte textuelle Form gebracht, durch die eine direkte Umsetzung in eine formale Spezifikation möglich ist.

Das vorliegende Kapitel gibt einen auf diese Arbeit zugeschnittenen Überblick über die semantische Basis von Focus und die verwendeten Beschreibungstechniken. Die wesentlichen semantischen Grundlagen ausgehend von den Konzepten der *Ströme* und der *stromverarbeitenden Funktionen*, den Möglichkeiten zur Einbeziehung von *Zeit* und der Beschreibung von *statischen* sowie von *mobilen, dynamischen Systemstrukturen* werden vorgestellt und soweit wie möglich nur informell erläutert, wobei auf weiterführende Literatur zu Erweiterungen und Alternativen hingewiesen wird. Diese zusammenfassende Darstellung liefert die Basis für die später erstellten Spezifikationen und wurde für Leser geschrieben, die mit Focus nicht oder nur wenig vertraut sind.

2.1 Was ist Focus?

Focus, siehe [BDD+93] und [BS98], ist ein allgemeiner Rahmen zur Entwicklung verteilter, reaktiver und interagierender Systeme. Focus kann überall dort eingesetzt werden, wo Systeme aus einer Menge räumlich oder konzeptionell verteilter Komponenten zusammengesetzt sind, die zueinander und zu ihrer Umgebung in Wechselwirkung stehen. Eine Entwicklung erfolgt in *Top-Down*-Vorgehensweise, wobei ein System ausgehend von einer ersten (informellen) Beschreibung der gestellten Anforderungen über mehrere Entwicklungsstufen hinweg bis zum gewünschten Detaillierungsgrad, möglicherweise einer Implementierung, entwickelt wird. Zur Formalisierung verschiedener Aspekte des Systems werden unterschiedliche *Beschreibungstechniken* eingesetzt. Die Semantik liefert eine mathematische Beschreibung dafür, wie einzelne Systemteile zueinander in Beziehung stehen, wodurch das Verhalten des Systems bzw. der Systemteile festgelegt wird.

Durch die Verknüpfung der *Komponenten* über gerichtete *Kanäle* wird die *Struktur* eines verteilten Systems festgelegt. Sowohl für Komponenten als auch für Systeme wird eine einheitliche Sichtweise eingenommen. Systeme werden ihrerseits als Komponenten mit Kanalverbindungen von bzw. zur *Umgebung* des Systems modelliert. Komponenten können interagieren oder unabhängig voneinander arbeiten, wobei die Art der Vernetzung abhängig von den bestehenden Kanalverbindungen ist. Bestehen zwischen Komponenten keine Kanalverbindungen, so sind sie unabhängig voneinander. Vor allem bei der Entwicklung komplexer Systeme müssen Komponenten separat entwickelt, zu einem Gesamtsystem komponiert oder in die Spezifikation eines bestehenden Systems integriert werden können. Focus unterstützt diese Vorgehensweise durch eine *modulare* Systementwicklung mit einer *kompositionalen* semantischen Basis.

Gemäß der Konzeption von Focus interagieren die Komponenten durch den asynchronen Austausch von Nachrichten über die bestehenden Kanalverbindungen. Das *beobachtbare Verhalten* eines Systems entspricht dem Nachrichtenfluß und wird festgelegt durch die Beziehungen zwischen den Nachrichten, die eine Komponente empfängt, und den Nachrichten, die sie verschickt. Auf ihre Eingaben nehmen Komponenten keinen Einfluß[1], während die Ausgaben ausschließlich durch die Komponente und gemäß dem spezifizierten Verhalten bestimmt werden. Die für eine Komponente definierten Ein- und Ausgabekanäle zusammen mit den zugeordneten Nachrichten bilden die *Schnittstelle* einer Komponente.

Focus stellt Richtlinien zur Entwicklung verteilter Systeme bereit. Diese Methodik *im Großen* umfaßt eine allgemeine Top-Down-Vorgehensweise, die Bereitstellung von Beschrei-

[1] Es sei denn, für die Komponente ist ein Rückkopplungskanal definiert; vgl. hierzu auch Abschnitt 2.5.

bungstechniken und einen Verfeinerungsbegriff, mit dem Spezifikationen zueinander in Beziehung gesetzt werden können. Methodische Hinweise *im Kleinen*, wie beispielsweise eine Unterstützung zur Spezifikationserstellung in einer ausgewählten Beschreibungstechnik, werden im allgemeinen nicht gegeben. Erste Arbeiten zu methodischem Vorgehen sind auf bestimmte Anwendungsbereiche oder spezielle Beschreibungstechniken zugeschnitten, siehe [HS96] oder [HS97] für mobile, dynamische Systeme oder die Modellierung der Betriebssystemkonzepte in der vorliegenden Arbeit.

2.2 Ströme

Wechselwirkungen der Komponenten untereinander bzw. zwischen System und Umgebung werden durch Nachrichtenaustausch beschrieben. Das grundlegende Konzept hierfür sind *Ströme*, die endliche und unendliche (vollständige) Sequenzen von Nachrichten repräsentieren und die über die Kanäle fließenden *Kommunikationsgeschichten* darstellen. Ausgehend von einer Nachrichtenmenge M ist die Menge der Ströme M^ω definiert durch die Menge aller endlichen (M^*) und unendlichen (M^∞) Sequenzen über M. Es gilt:

$$M^\omega = M^* \cup M^\infty$$

Zur Festlegung der Menge M und somit für die Auswahl der Nachrichten, richtet sich der Entwickler nach dem für die Spezifikation gewählten Abstraktionsgrad; Beispiele liefern die Spezifikationen der Kapitel 4 bis 7. Alle dabei gewählten abstrakten Nachrichten könnten in späteren Entwicklungsphasen beispielsweise durch die für eine konkrete Hardwarekonfiguration gültigen speziellen Signale ersetzt (verfeinert) werden.

Für den Umgang mit Strömen sind spezielle Operationen[2] und Notationen definiert:

$\langle\rangle$	bezeichnet den *leeren Strom*, also das einzige Element in M^ω, das keine Nachrichten enthält.
$\langle a_1, \ldots, a_n \rangle$	bezeichnet für $a_1, \ldots, a_n \in M$ den endlichen Strom, der mit a_1 beginnt und mit a_n endet.
$\circ : M^\omega \times M^\omega \to M^\omega$	bezeichnet die Konkatenation zweier Ströme. $s \circ t$ liefert den Strom, der mit s beginnt und mit t fortgesetzt wird, sofern s endlich ist. Für $s \in M^\infty$ gilt $s \circ t = s$.
$ft : M^\omega \to M$	liefert das erste Element eines Stromes, und es gilt $ft.\langle\rangle = \langle\rangle$.
$rt : M^\omega \to M^\omega$	liefert den Strom, der sich ergibt, wenn dessen erstes Element entfernt wird. Es gilt: $rt.\langle\rangle = \langle\rangle$.
$.[k] \; : \; (I\!N \times M^*) \to M$	liefert das $k$te Element eines endlichen Stroms. Für $s \in M^*$ gilt $s[1] = ft.s$ und $s[k] = undef.$, falls $k > \#s$.

[2]Für eine zusammenfassende Darstellung siehe [BS98]. Insbesondere können die Operationen in natürlicher Weise auch auf Tupel von Strömen erweitert werden.

$\sqsubseteq : M^\omega \times M^\omega \to I\!B$ bezeichnet die Präfixrelation. $s \sqsubseteq t$ für $s, t \in M^\omega$ ist gültig, falls s ein Anfangsstück (Präfix) von Strom t oder identisch zu Strom t ist. Es gilt:

$$\forall s, t \in M^\omega : s \sqsubseteq t \quad \Leftrightarrow \quad \exists r \in M^\omega : s \circ r = t$$

$\# : M^\omega \to I\!N_0$ liefert die *Länge* eines endlichen Stromes oder ∞, falls der Strom unendlich ist[3].

$© : \mathcal{P}(M) \to M^\omega$ Mit $M' \subseteq M$ liefert $M' © s$ den Teilstrom von $s \in M^\omega$, der ausschließlich Elemente aus M' enthält[4].

Die Relation $\sqsubseteq$ liefert eine partielle Ordnung auf Strömen. Zusammen mit $\sqsubseteq$ bilden die Ströme eine vollständige Halbordnung (*cpo*) mit $\langle\rangle$ als kleinstem Element und den unendlichen Strömen als Abschlüssen. $\sqsubseteq$ wird verwendet, um Ströme hinsichtlich der Information, die sie enthalten, zu vergleichen; bei $s \sqsubseteq t$ enthält t mindestens soviel Information wie s.

Die Eigenschaften, die für das Systemverhalten gefordert werden, werden mittels prädikatenlogischer Formeln als Aussagen über die Systemabläufe präzisiert. Systementwicklungen können alternativ zu den stromverarbeitenden Funktionen, siehe die Abschnitte 2.3ff, auch als *Trace-* (oder Spur-) Spezifikationen erstellt werden, vergleiche u.a. [Web92].

2.3 Stromverarbeitende Funktionen

Um die Interaktionen der Komponenten untereinander bzw. zwischen System und Umgebung zu beschreiben, werden *stromverarbeitende Funktionen* eingesetzt. Erste grundlegende Arbeiten über stromverarbeitende Funktionen sowie den Einsatz von Gleichungssystemen zur Spezifikation verteilter Systeme sind in [Kah74], [BA81] sowie [Bro87] zu finden. Wir verwenden in allen gezeigten Spezifikationen den konstruktiven, operationalen Stil, da er der algorithmischen Sichtweise in Betriebssystemen nahekommt und in gewisser Weise mit funktionaler Programmierung vergleichbar ist.

Das Verhalten des Systems bzw. der Komponenten[5] wird durch Abbildungen von Tupel von Eingabe- auf Tupel von Ausgabeströmen beschrieben. Wir gehen von der Vorstellung aus, daß eine Komponente Nachrichten von ihren Eingabekanälen liest und in Abhängigkeit von diesen Nachrichten Ausgaben produziert. Für jede Komponente werden die Ein- und Ausgabekanäle, die durch eindeutige Bezeichner identifiziert werden, sowie die Nachrichten, die über die Kanäle fließen dürfen, festgelegt. Mit diesen Angaben ist die *Schnittstelle* einer Komponente definiert. Durch die Schnittstelle wird für die Lebensdauer einer Komponente festgelegt, von welchen Kanälen sie lesen und auf welche sie schreiben darf. Wir betrachten sogenannte *statische* Systemstrukturen. Veränderungen der Schnittstelle einer Komponente oder des strukturellen Aufbaus eines Systems können mittels einer *Verfeinerung* beim

[3] $I\!N_0$ steht für die Menge der natürlichen Zahlen einschließlich der 0.

[4] $\mathcal{P}(M)$ steht für die Potenzmenge der Menge M, und © wird als *Filter*funktion bezeichnet.

[5] Im folgenden sprechen wir nur noch von *Komponenten*. Die Aussagen gelten entsprechend für *Systeme* – es sei denn, ein Unterschied wird explizit genannt.

Übergang von einer Entwicklungsstufe zur nächsten bewältigt werden; vergleiche hierzu Abschnitt 2.5. Die Modellierung von Schnittstellenveränderungen während eines Systemablaufs ist im erweiterten sematischen Modell möglich, siehe Abschnitt 2.6.

Für eine Komponente K mit n Eingabe- und m Ausgabekanälen und hierfür festgelegten Nachrichtentypen $InMsg_1, \ldots, InMsg_n$ für die Eingabe- bzw. $OutMsg_1, \ldots, OutMsg_m$ für die Ausgabekanäle wird ein mögliches Verhalten durch eine Funktion f_K vom Typ

$$f_K \;\in\; InMsg_1^\omega \times \cdots \times InMsg_n^\omega \;\rightarrow\; OutMsg_1^\omega \times \cdots \times OutMsg_m^\omega \tag{2.1}$$

festgelegt, die n-Tupel von Strömen auf m-Tupel von Strömen abbildet. Um die Zuordnung von Kanälen zu ihren Strömen zu verdeutlichen, werden sogenannte *benannte* stromverarbeitende Funktionen verwendet. Damit wird für die Mengen $IN = \{In_1, \ldots, In_n\}$ von Bezeichnern der Eingabekanäle und $OUT = \{Out_1, \ldots, Out_m\}$ von Bezeichnern der Ausgabekanäle das Verhalten einer Komponente festgelegt durch

$$f_K \;\in\; (IN \mapsto \prod_{i=1}^{n} InMsg_i^\omega) \;\longrightarrow\; (OUT \mapsto \prod_{j=1}^{m} OutMsg_j^\omega) \tag{2.2}$$

Das kartesische Produkt $\prod$ ordnet jedem Kanalbezeichner eine vollständige Kommunikationsgeschichte über der Menge zu, die die für den Kanal definierten Nachrichten enthält.

Für alle Spezifikationen setzen wir voraus, daß sich die Umgebung einer Komponente K bzgl. der Eingaben typkorrekt verhält. K erhält nur solche Nachrichten, die für den Kanal definiert sind. Zur Spezifikation statischer Systemstrukturen können die stromverarbeitenden Funktionen gemäß (2.1) und die benannten stromverarbeitenden Funktionen gemäß (2.2) gleichermaßen verwendet werden.

Für stromverarbeitende Funktionen werden folgende semantische Eigenschaften gefordert: Die Funktionen sind *monoton*[6] bzgl. der Präfixordnung $\sqsubseteq$. Dies beschreibt die Kausalität zwischen den Ein- und Ausgaben. Jedes Präfix enthält sämtliche Information, die aus dem bisherigen Verhalten entnommen werden kann und über das weitere Verhalten entscheidet; bereits ausgegebene Nachrichten können nicht mehr zurückgenommen werden. Monotone Funktionen besitzen einen eindeutigen kleinsten *Fixpunkt*. Damit ist sichergestellt, daß eine Komponente eine *Berechnung*, die zu einem eindeutigen Ergebnis führt, vollständig ausführt. Die Funktionen sind *stetig*[7], wodurch der kleinste Fixpunkt eines vollständigen (unendlichen) Systemablaufs aus seinen endlichen Approximationen *konstruktiv* berechnet werden kann. Wird eine Spezifikation mittels rekursiver Funktionsgleichungen und unter Verwendung der im Abschnitt 2.2 angegebenen Operationen in dem von uns geforderten operationalen Stil mittels der Schemata erstellt, ist sichergestellt, daß die Formalisierung stetige Funktionen liefert.

[6] $f : In^\omega \mapsto Out^\omega$ ist *monoton*, wenn für $s, t \in In^\omega$ gilt: $s \sqsubseteq t \;\Rightarrow\; f(s) \sqsubseteq f(t)$.

[7] Eine monotone Funktion $f : In^\omega \mapsto Out^\omega$ ist *stetig*, wenn gilt: $\sqcup_i \; \{f(s_i) | (s_i | i \in I\!N)$ Kette aus $In^\omega\} = f(\sqcup_i s_i)$. $\sqcup_i s_i$ steht für die kleinste obere Schranke.

Das Verhalten einer Komponente wird durch eine Menge stromverarbeitender Funktionen beschrieben, die durch prädikatenlogische Formeln charakterisiert wird. Sei S_K die FOCUS-Spezifikation einer Komponente K. Die *Semantik* von S_K wird mit

$$[[S_K]] \;=\; \{f \mid P_{S_K}.f \;\wedge\; f \text{ stetig}\}$$

bestimmt. Hierbei verfüge K über die vorher festgelegte Schnittstelle, so daß die in $[[S_K]]$ verwendeten Funktionen von dem mit (2.1) bzw. (2.2) vorgegebenen Typ sind. Das Verhalten von K wird durch alle (benannten) stromverarbeitenden Funktionen f beschrieben, die das Prädikat P_{S_K} erfüllen, und entspricht allen Paaren von Ein- und Ausgabestromtupeln, die gemäß P_{S_K} gültig sind. Da die Menge $[[S_K]]$ mehrere Funktionen enthalten kann, die P_{S_K} erfüllen, ist die Beschreibung von *Nichtdeterminismus* möglich. Als Bezeichner für eine Spezifikation verwenden wir meistens den Komponentenbezeichner und schreiben $[[K]]$.

Für weitere Erläuterungen, für detaillierte Beschreibungen der Semantik und der Eigenschaften, die (benannte) stromverarbeitende Funktionen erfüllen, verweisen wir auf weiterführende Literatur, wie beispielsweise [BS98] oder [BDD+93]. Da in der vorliegenden Arbeit alle Spezifikationen im funktionalen Stil erstellt werden, gehen wir im folgenden nur noch auf diesen ein. Zur Erstellung von Spezifikationen im funktionalen Stil bietet FOCUS eine Reihe graphischer und anwendungsnaher Beschreibungstechniken, wie beispielsweise Automatendiagramme ([GKRB96]), Tabellen ([Spi94]) oder auch EET's, vergleiche [BHS96], [HSE97] und Abschnitt 2.7.

Spezifikationsschemata

Zur Erstellung von Spezifikationen im funktionalen Stil gibt es im wesentlichen zwei Möglichkeiten: In einer abstrakteren Sicht werden vollständige Ein- und Ausgabeströme *deskriptiv* charakterisiert. Hierbei werden im allgemeinen keine Aussagen darüber gemacht, wie Ausgaben konstruktiv aus den Eingabeströmen *berechnet* werden. Wir werden im Gegensatz dazu einen *konstruktiven* Stil verwenden, der für unsere Anwendung geeigneter und insgesamt leichter nachvollziehbar erscheint. Die Spezifikationen werden als Gleichungssysteme über stromverarbeitenden Funktionen angegeben. Im Rahmen dieser näher an der Ausführung von Berechnungen orientierten Sicht wird das Verhalten einer Komponente beschrieben, indem die Eingaben schrittweise verarbeitet und die Ausgabeströme sukzessive *aufgebaut* werden. Im folgenden werden wir Spezifikationsmuster für diesen Spezifikationsstil festlegen. Da wir ausschließlich benannte stromverarbeitende Funktionen verwenden, werden die Schemata nur für diese Variante angegeben.

Eine Komponente K verfüge über die bereits für (2.1) und (2.2) auf Seite 15 festgelegte Schnittstelle. Das Verhalten von K wird durch $[[K]] = \{f \mid P_K.f\}$ festgelegt und alle f haben den mit (2.2) festgelegten Typ. Mit dem Schema[8]

$$f(\{In_1 \mapsto X_1, \ldots, In_n \mapsto X_n\} \circ s) \;=\; \{Out_1 \mapsto Y_1, \ldots, Out_m \mapsto Y_m\} \circ f(s) \tag{2.3}$$

[8]Die Fixpunktberechnung ist hierbei gewährleistet. Zur semantischen Fundierung siehe u.a. [SS95].

wird folgende Situation beschrieben:

> Empfängt K die Nachrichten $X_1 \in InMsg_1$ bis $X_n \in InMsg_n$ über die Kanäle In_1 bis In_n, so sendet K die Nachrichten $Y_1 \in OutMsg_1$ bis $Y_m \in OutMsg_m$ über die Kanäle Out_1 bis Out_m.

Stromverarbeitende Funktionen können einen zusätzlichen Parameter erhalten, der zur Modellierung von *Zuständen* eingesetzt wird. Für eine allgemeine Beschreibung mit einer Reihe von Beispielen verweisen wir auf [BS98]. Für eine Komponente K mit der bekannten Schnittstelle und einer Zustandsmenge $State_K$ ergibt sich der Funktionstyp:

$$f \;\in\; State_K \;\rightarrow\; ((IN \mapsto \prod_{i=1}^{n} InMsg_i^{\omega}) \;\longrightarrow\; (OUT \mapsto \prod_{j=1}^{m} OutMsg_j^{\omega})) \qquad (2.4)$$

Das Schema für zustandsbasierte Spezifikationen lautet

$$f(z)(\{In_1 \mapsto X_1, \ldots, In_n \mapsto X_n\} \circ s) \;=\; \{Out_1 \mapsto Y_1, \ldots, Out_m \mapsto Y_m\} \circ f(z')(s) \quad (2.5)$$

und beschreibt das folgende Verhalten

> Empfängt K im Zustand $z \in State_K$ die Nachrichten $X_1 \in InMsg_1$ bis $X_n \in InMsg_n$ über die Kanäle In_1 bis In_n, so sendet K die Nachrichten $Y_1 \in OutMsg_1$ bis $Y_m \in OutMsg_m$ über die Kanäle Out_1 bis Out_m und geht in den Folgezustand $z' \in State_K$ über.

Der Zustandsparameter kann in unterschiedlicher Bedeutung verwendet werden: Als *Datenzustand*, wie bei der Modellierung einer Warteschlange, siehe Abschnitt 4.3.7, oder als *Kontrollzustand*, wie die Zustände eines Prozesses, siehe Abschnitt 4.3.3. Zur Verwendung endlicher, statischer Zustandsräume und deren Einbettung in HOLCF und FOCUS siehe [Ohe95]. In den folgenden Kapiteln werden wir Spezifikationen sowohl mit als auch ohne Zustandsparameter erstellen und Zustände in beiden Ausprägungen verwenden. Bei der Festlegung einer Zustandsmenge werden in der Spezifikation entsprechend zur Festlegung des Nachrichtentyps mehr oder weniger abstrakte Bezeichner verwendet.

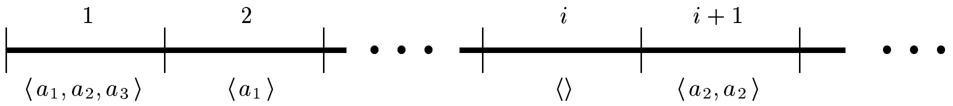
Bei der Erstellung einer formalen Spezifikation werden die Anforderungen, die an eine Komponente gestellt werden, durch den konsequenten Einsatz derartiger Schemata in Funktionsgleichungen umgesetzt. Diese Gleichungen sind konjunktiv verknüpft und definieren so die Funktionen, die der Spezifikation genügen. Der Detaillierungsgrad, in dem das Verhalten einer Komponente spezifiziert wird, hängt von den geforderten Eigenschaften ab. Das Konzept der *Unterspezifikation* von FOCUS erlaubt hier, daß das Verhalten nicht notwendigerweise für alle mit den definierten Eingabenachrichten konstruierbaren Eingabeströme definiert werden muß. Für weitere Erläuterungen siehe [BDD+93] und [BS98].

2.4 Modellierung von Zeit

Zur expliziten Formulierung von Zeitabhängigkeiten beim Austausch von Nachrichten wird das bisher erklärte semantische Modell erweitert. Es wird vorausgesetzt, daß für ein verteiltes System eine *globale Uhr* definiert ist, die das Voranschreiten der Zeit anzeigt. In einem Zeitabschnitt treten immer endlich viele Nachrichten auf. Ströme werden über endliche Sequenzen gebildet[9], und ein unendlicher *gezeiteter* Strom stellt eine vollständige Kommunikationsgeschichte dar. Wir legen fest:

$[M^*]$	ist die Menge der vollständigen gezeiteten Ströme über M.
$(M^*)^*$	ist die Menge aller endlichen gezeiteten Ströme über M.

Für M mit $\{a_1, a_2, a_3\} \subseteq M$ erhalten wir beispielsweise mit



einen gezeiteten Strom, in dem die Nachrichten a_1, a_2 und a_3 im ersten Zeitintervall, im zweiten Intervall nur a_1, im i-ten Zeitintervall die leere Sequenz, also keine Nachricht, und im $(i+1)$-ten Zeitintervall a_2 doppelt auftreten.

Für den Umgang mit gezeiteten Strömen sind spezielle Operationen definiert. Wir nennen hier nur $s\,|_j$, mit dem das Präfix von $s \in [M^*]$ inklusive des j-ten Zeitintervalls bestimmt wird, und der auch für benannte Stromtupel definiert ist. Für eine präzise Definition und weitere Operationen für gezeitete Ströme siehe [BS98].

Pulsgetriebene stromverarbeitende Funktionen

Die Einbeziehung gezeiteter Ströme in das Konzept der stromverarbeitenden Funktionen führt zu *pulsgetriebenen Funktionen*. Diese verarbeiten Ströme von Sequenzen und mit ihnen werden Kausalitäten zwischen den Ein- und Ausgaben sowie explizites Zeitverhalten modelliert. Für eine Komponente K mit der auf Seite 15 festgelegten Schnittstelle gilt:

$$f \;:\; (IN \mapsto \prod_{i=1}^{n} [InMsg_i^*]) \;\longrightarrow\; (OUT \mapsto \prod_{j=1}^{m} [OutMsg_j^*]) \tag{2.6}$$

Eine benannte, stromverarbeitende Funktion f gemäß (2.6) ist *stark pulsgetrieben*, wenn für alle Eingaben $\alpha, \beta \;:\; IN \mapsto \prod_{i=1}^{n} \; [InMsg_i^*]$ gilt:

$$\alpha|_j \;=\; \beta|_j \quad\Rightarrow\quad f(\alpha)|_{j+1} \;=\; f(\beta)|_{j+1}$$

Die starke Pulsgetriebenheit entspricht einer strengeren und auf das Voranschreiten der Zeit zugeschnittenen Form der Monotonie. Die Eingaben bis zum Zeitpunkt j bestimmen

[9]Alternativ wird oft das spezielle (von allen übrigen Nachrichten verschiedene) Zeichen $\surd$ („Tick") verwendet, wobei $\surd$ den Übergang zum nächsten Zeitintervall markiert. Ströme werden hierbei über der Menge $M_{\surd} = M \cup \{\surd\}$ gebildet. Vergleiche hierzu [BDD+93] oder auch [Fuc94].

vollständig die Ausgaben bis zum Zeitpunkt $j+1$. Eingaben werden in jedem Zeitintervall verarbeitet, und Ausgaben hängen insbesondere nicht von zukünftigen Eingaben ab. Mittels stark pulsgetriebener Funktionen werden Komponenten modelliert, die zur Verarbeitung der Eingaben eines Zeitintervalls und der Erzeugung der nächsten Ausgabe mindestens eine Zeiteinheit benötigen. Mit dieser Eigenschaft ist die Existenz eines eindeutigen Fixpunktes gesichert. Mit *schwach pulsgetriebenen*[10] Funktionen werden Komponenten modelliert, die zur Verarbeitung der Eingaben keine Zeit verbrauchen. Auch pulsgetriebene Funktionen können um einen Zustandsparameter erweitert werden. Der erweiterte Typ wird gemäß (2.6) und entsprechend zu (2.4) bestimmt. Die Semantik $[[S]]$ einer Spezifikation S ergibt sich mittels prädikatenlogischer Formeln analog zu Abschnitt 2.3 und Seite 16. Wir verwenden in unseren Modellierungen ausschließlich stark pulsgetriebene Funktionen.

Für das gezeitete semantische Modell sind einige Varianten definiert. Wird in jedem Zeitintervall genau eine Nachricht verarbeitet, so sprechen wir von *synchronen* Funktionen. Wird in jedem Zeitintervall jeweils eine vollständige Sequenz gelesen bzw. gesendet wird, nennen wir die Funktionen *getaktet*. Mit *zeitunabhängige* Funktionen werden an eine Komponente explizit keine Anforderungen an das Zeitverhalten gestellt. Hierbei wird gefordert, daß von der Zeitinformation vollständig abstrahiert werden kann. Die Ausgaben sind unabhängig vom Zeitverhalten der Eingaben, und es werden keine Zeitbedingungen an die Ausgaben gestellt. Für detailliertere Angaben zu pulsgetriebenen Funktionen und für die Formalisierung der hier genannten Eigenschaften wird auf die mehrfach genannte Literatur zu FOCUS verwiesen. In [Bro97] werden Varianten des gezeiteten Modells vorgestellt, insbesondere *kontinuierliche* und *dichte* Zeitmodelle.

Spezifikationsschemata und spezielle Konventionen

Zur Erstellung von Spezifikationen legen wir, entsprechend zu unserer bisher gezeigten Vorgehensweise, ein Schema fest und führen spezielle Konventionen ein, die für alle später gezeigten Spezifikationen gelten. Für das Schema sei K eine Komponente mit der auf Seite 15 festgelegten Schnittstelle. Das Verhalten von K sei durch $[[K]] = \{f \mid P_K.f\}$ festgelegt, wobei f vom durch (2.6) festgelegten Typ (bzw. zustandsbasiert mit $State_K$) und pulsgetrieben ist.

Ein Verhalten sei textuell beschrieben durch

> Erhält K im Zustand $z \in State_K$ die Sequenzen $\langle X_{1_1}, \ldots, X_{1_n} \rangle \in InMsg_1^*$ bis $\langle X_{n_1} \ldots, X_{n_n} \rangle \in InMsg_n^*$ über die Kanäle In_1 bis In_n, so sendet K die Sequenzen $\langle Y_{1_1}, \ldots, Y_{1_m} \rangle \in OutMsg_1^*$ bis $\langle Y_{m_1}, \ldots, Y_{m_m} \rangle \in OutMsg_m^*$ über die Kanäle Out_1 bis Out_m und geht in den Folgezustand $z' \in State_K$ über.

[10]Schwache Pulsgetriebenheit ist definiert mit $\alpha|_j = \beta|_j \;\Rightarrow\; f(\alpha)|_j = f(\beta)|_j$. Die Komposition von schwach und stark pulsgetriebenen Funktionen führt zu starker Pulsgetriebenheit.

Hierfür legen wir folgende schematische Umsetzung fest:

$$\begin{aligned} f(z)(\{In_1 \mapsto \langle X_{1_1}, \ldots, X_{1_n} \rangle, \ldots, In_n \mapsto \langle X_{n_1} \ldots, X_{n_n} \rangle\} \circ s) \;&= \\ \{Out_1 \mapsto \langle Y_{1_1}, \ldots, Y_{1_m} \rangle, \ldots, Out_m \mapsto \langle Y_{m_1}, \ldots, Y_{m_m} \rangle\} \circ f(z')(s) & \qquad (2.7) \end{aligned}$$

Der Zustandsparameter ist optional, das Schema gilt ohne Parameter entsprechend. Für zeitunabhängige Funktionen können die Schemata (2.3) bzw. (2.5) verwendet werden.

Wir fordern für die in den späteren Kapiteln erstellten Spezifikationen, daß die stromverarbeitenden Funktionen stets *stark pulsgetrieben* sind. Es ist somit gewährleistet, daß die gesamte Ausgabe relativ zur Eingabe um (mindestens) einen Zeittakt verzögert wird.

Im *getakteten* Format werden immer alle Eingabesequenzen in einem Zeitintervall gelesen. Da es nicht immer notwendig (oder sogar erwünscht) ist, jeweils alle Eingabekanäle explizit zu behandeln, führen wir folgende Konvention ein: Falls in einer Funktionsgleichung nur einige Eingabekanäle der Schnittstelle mit Eingabesequenzen aufgeführt sind, so bedeutet dies, daß hier alle anliegenden Nachrichten verarbeitet werden. Auf die Eingaben, die nicht explizit genannt werden, wird nicht reagiert, sie gehen verloren. Über alle Ausgabekanäle, die nicht explizit aufgeführt sind, wird die leere Sequenz ausgegeben. Diese Konvention hat zur Folge, daß immer alle Eingabemuster, die explizit behandelt werden sollen, durch eine entsprechende Funktionsgleichung formalisiert werden müssen. Für eine Funktion f vom Typ (2.6) wird eine Gleichung

$$f(\{In_i \mapsto \langle X \rangle\} \circ s) \;=\; \{Out_j \mapsto \langle Y \rangle\} \circ f(s) \qquad (2.8)$$

abkürzend verwendet für

$$\begin{aligned} \forall s_k \in InMsg_k^* \;:\; & f(\{In_1 \mapsto s_1, \ldots, In_i \mapsto \langle X \rangle, \ldots, In_n \mapsto s_n\} \circ s) \\ & = \{Out_1 \mapsto \langle\rangle, \ldots, Out_j \mapsto \langle Y \rangle, \ldots, Out_m \mapsto \langle\rangle\} \circ f(s) \end{aligned}$$

2.5 Verteilte Systeme und Verfeinerung

Im folgenden nennen wir eine Komponente, die selbst nicht als verteiltes System beschrieben wird, *Basiskomponente*, und ein verteiltes System bezeichnen wir auch als *Netz*. Focus ist *hierarchisch* konzipiert, weshalb auch Netze durch stromverarbeitende Funktionen modelliert werden. Ein System kann in zwei Sichten beschrieben werden:

- Die *Black-Box*-Sicht zeigt das System als Einheit ohne Berücksichtigung des strukturellen Aufbaus.
- Die *Glass-Box*-Sicht berücksichtigt zusätzlich die Vernetzung der Komponenten.

In der Glass-Box-Sicht wird das Verhalten des Systems durch das Zusammenspiel der Komponenten bestimmt. Die Vernetzung der Komponenten ergibt sich aus den Kanalverbindungen der Komponenten, und alle Kanäle, die eine Komponente mit der Umgebung verbinden, liefern die Verbindung des Systems zur Umgebung. Auf dieser Basis ist in Focus der Begriff des *beobachtbaren Verhaltens* geprägt: Es sind ausschließlich die Nachrichtenflüsse beobachtbar, die von der Umgebung gesendet oder empfangen werden.

Zur Komposition stromverarbeitender Funktionen ist ein allgemeiner Kompositionsoperator $\otimes$ definiert. Spezielle Formen sind die *sequentielle* sowie die *parallele* Verknüpfung von Komponenten. Komponenten arbeiten sequentiell, wenn die Ausgabenachrichten einer Komponente die Eingabe für eine andere Komponente bilden. Parallel komponierte Komponenten arbeiten unabhängig voneinander. Im Fall einer *Rückkopplung*, schickt eine Komponente Ausgabenachrichten an sich selbst zurück. Die Spezifikation eines verteilten Systems V erfolgt durch die Verknüpfung der Spezifikationen seiner Komponenten. V sei aus den Komponenten A und B zusammengesetzt. $[[A]]$ bzw. $[[B]]$ bezeichnen jeweils die Semantik dieser Komponenten. Für V mit $A \otimes B$ legen wir als Semantik fest:

$$[[V]] \;=\; \{f \mid f = g \otimes h \;\wedge\; g \in [[A]] \wedge\; h \in [[B]]\}$$

Für die Modellierung verteilter Systeme geben wir hier keine weiteren Notationen und Schemata an, da diese mittels Andl, siehe Abschnitt 2.8, spezifiziert werden.

Für $\otimes$ ist die semantische Eigenschaft der *Kompositionalität* gewährleistet, so daß die Basiskomponenten eines Netzes separat (*modular*) spezifiziert werden können. Zur detaillierten Beschreibung des Kompositionsoperators, zur Charakterisierung der speziellen Kompositionsformen und deren semantischen Eigenschaften verweisen wir auf die bereits mehrfach angegebene Literatur [BS98], [BDD$^+$93] sowie auch auf [FS93].

Verfeinerung

Eine formale Top-Down-Systementwicklung wird in Focus mittels schrittweiser Verfeinerung durchgeführt, wobei für die Durchführung der Entwicklungsschritte ein kompositionales Verfeinerungskonzept und Verfeinerungskalküle zur Verfügung stehen, siehe [Bro92a], [Bro92b] und [BS98]. Das Verhalten von Komponenten wird durch Mengen von Funktionen, die prädikativ spezifiziert werden, dargestellt. Auf dieser Basis kann die Gültigkeit der Verfeinerungsschritte mittels logischer Implikation nachgewiesen werden. Die drei wesentlichen Verfeinerungskonzepte umfassen die Verfeinerung von:

Verhalten: zur Reduktion von Unterspezifikation;

Schnittstelle: zur Weiterentwicklung und Konkretisierung einer Schnittstelle;

Struktur: zur Entwicklung des strukturellen Aufbaus eines Systems.

Da diese Konzepte in der vorliegenden Arbeit nur implizit (bei der Erzeugung von Komponenten, siehe Abschnitt 2.6) verwendet werden, verweisen wir auf [BS98], [Bro92a] und [Bro92b]. Für erste methodische Hilfestellungen bei der Durchführung von Beweisen, den Einsatz der Konzepte bei der Behandlung einiger Beispiele sowie einer Anbindung an das interaktive Beweiswerkzeug *Isabelle* siehe die Arbeiten [San96], [Slo97], [Hin97], [Bre97], [Bro96] oder [Stø96].

2.6 Mobile und dynamische Systeme

Die bisher erläuterten semantischen Konzepte sind für Systeme mit statischer Struktur konzipiert. Mobile, dynamische Systeme in FOCUS sind dadurch charakterisiert, daß sich ihre Systemstruktur, ausgelöst durch Interaktionen und Berechnungen, während eines Systemablaufs verändern kann. Mit den Begriffen *mobil* und *dynamisch* werden die möglichen Veränderungen der Systemstruktur während eines Systemablaufs beschrieben.

1. Die Vernetzung der Komponenten kann sich ändern. Dies kann durch das Umlenken bestehender Verbindungen oder durch das Erzeugen neuer bzw. das Löschen bestehender Kommunikationsverbindungen erreicht werden. Systeme, für die derartige Veränderungen charakteristisch sind, werden in FOCUS *mobile Systeme* genannt.

2. Die Anzahl der im System existierenden Komponenten kann sich ändern, indem neue Komponenten dynamisch erzeugt und in das System integriert oder aktuell existierende Komponenten aus dem System gelöscht werden. Systeme, für die diese Veränderungen charakteristisch sind, werden in FOCUS *dynamische Systeme* genannt.

Mit dem im folgenden erklärten, erweiterten semantischen Modell kann sowohl mobiles als auch dynamisches Verhalten modelliert werden.

Semantische Konzepte und wichtige Begriffe

Um die mit 1. und 2. genannten Veränderungen eines Systems in FOCUS beschreiben zu können, wurde der klassische, bisher beschriebene FOCUS-Ansatz erweitert. Als ersten Ansatz enthält [Bro95a] einen logischen Kalkül, in dem auf der Basis hierarchischer Gleichungssysteme dynamische Änderungen der Systemstrukturen modelliert werden können. Es werden Regeln angegeben, die korrekte Substitutionen in den Gleichungen festlegen. Das im folgenden erklärte semantische Modell ist motiviert durch den π-Kalkül, siehe [MPW92a] und [MPW92b], neben dem Actor-Modell, siehe [AMST92] und [HBS73], oder auch dem Higher Order CCS, siehe [Tho89], einer der klassischen Ansätze zur Beschreibung von Mobilität. Erste Arbeiten zu einem auf den pulsgetriebenen stromverarbeitenden Funktionen basierenden semantischen Modell sind in [Gro94] enthalten. Detailliert wird das denotationelle Modell für gezeitete, nichtdeterministische, mobile Systeme in [GS96b], [GS96c] und [GS96a] beschrieben. In [Gro96a] und [Gro96b] wird das Modell um die Erzeugung von Komponenten erweitert. Eine Anleitung für den Umgang mit diesem Modell und die Modellierung des Löschens einer Komponente werden in [HS96] präsentiert sowie an einem Beispiel veranschaulicht. In [HS97] werden eine komprimierte Darstellung dieser Anleitung sowie ein Teil der im nächsten Abschnitt entwickelten Spezifikationsmuster anhand eines Beispiels präsentiert. Im folgenden geben wir eine zusammenfassende Erläuterung der semantischen Begriffe. Als Anleitung für die Verwendung des beschriebenen semantischen Konzepts geben wir Spezifikationsschemata an.

Das semantische Modell zur Beschreibung mobiler, dynamischer Systeme steht in zwei Varianten zur Verfügung. Im Modell mit *point-to-point*-Kommunikation, siehe [GS96c], ist sichergestellt, daß, wie auch im statischen Modell, ein Kanal ch jeweils genau zwei Komponenten bzw. einer Komponente und seiner Umgebung zur Verfügung steht: eine Komponente schreibt auf ch, die andere liest von ihm. Eine Verallgemeinerung dieser Kommunikationsform wird mit der *many-to-many*-Kommunikation, siehe [GS96b], vorgestellt, mit der auch $n \geq 1$ Komponenten schreibend und $m \geq 1$ Komponenten lesend auf einen Kanal zugreifen können. Für die Modellierung der Betriebssystemkonzepte verwenden wir ausschließlich die Variante mit point-to-point-Kommunikation. In [Hin97] wird beispielsweise das Modell mit many-to-many-Kommunikation verwendet, um die dynamische Prozeßgenerierung in der Spezifikationssprache SDL, siehe [IT93], zu modellieren.

In Focus werden Kanäle durch eindeutige Bezeichner mit entsprechender Typinformation identifiziert. Mit der Schnittstelle wird festgelegt, ob die Komponente von einem Kanal lesen, oder ob sie auf ihn schreiben darf. Gehört ein im System definierter Kanal nicht zur Schnittstelle einer Komponente, so darf diese den Kanal nicht nutzen. Komponenten verfügen über *Zugriffsrechte* (Lese- oder Schreibrecht) zu Kanälen. Zur Beschreibung dieser Rechte werden im erweiterten semantischen Modell sogenannte *Ports* eingeführt. Ports sind Kanalbezeichner, die zusätzlich mit einem Lese- (?) oder Schreibrecht (!) versehen sind. Sei N die Menge der im System definierten Kanalbezeichner. Für einen Kanal $ch \in N$ wird das Leserecht durch den Port $?ch$ und das Schreibrecht durch $!ch$ identifiziert. Eine wichtige Erweiterung des Modells besteht darin, daß über die Kanäle neben den Nachrichten auch Ports verschickt werden dürfen. Vollständige Abläufe mobiler, dynamischer Systeme können in Phasen aufgeteilt werden. Eine *Phase* charakterisiert einen *Schnappschuß* auf das System, in dem sich das System wie ein statisches System verhält.

Mit der auf Seite 22 in Punkt 1. gegebenen Charakterisierung ergibt sich, daß sich die Schnittstelle einer Komponente K im Gegensatz zu statischen Systemen während eines Systemablaufs verändern kann. Damit verändern sich die für K definierten Mengen der aktuell nutzbaren Ein- bzw. Ausgabekanäle. Um neue Kanäle erzeugen zu können, verfügt K zusätzlich über eine Menge *privater* Kanäle. Zu Beginn des Systemablaufs sind für K die Mengen I_K bzw. O_K der initialen Eingabe- bzw. Ausgabekanäle und die Menge P_K (mit $(I_K \cup O_K) \cap P_K = \emptyset$) der privaten Kanäle als *initiale Vernetzung* festgelegt. Um sicherzustellen, daß Komponenten nur Kanäle mit unterschiedlichen Kanalbezeichnern erzeugen, wird gefordert, daß die Mengen P_{K_i} für alle Komponenten K_i disjunkt sind.

Neben der initialen Vernetzung sind Mengen definiert, die die aktuelle Rechtevergabe an Kanälen regeln. Für eine Komponente K definiert ap_K die Menge der *aktiven Ports*, die die aktuelle Schnittstelle von K bestimmt. Für einen Kanal in_K, von dem K aktuell lesen kann, und einen Kanal out_K, auf den K aktuell schreiben darf, gilt $\{?in_K, !out_K\} \subseteq ap_K$. Die Menge pp_K der *passiven Ports* legt die privaten Kanäle von K fest. K darf private Kanäle aktuell zwar weder lesend noch schreibend nutzen, kann jedoch aus dieser Menge Ports entnehmen, um diese zu versenden und so neue Kanalverbindungen zu erzeugen. Die Mengen ap_K und pp_K sind im semantischen Modell inkrementell definiert und ändern sich, sobald K einen Port empfängt oder versendet. Ein aktiver Port $!out_K$ wird beispielsweise wieder passiv, wenn K über einen ihrer Eingabekanäle das komplementäre Recht zu dem Kanal – hier $?out_K$ – empfängt. In diesem Fall werden beide Ports $?out_K$ und $!out_K$ – hierfür schreiben wir abkürzend $?!out_K$ – in die Menge pp_K eingefügt. Die Veränderungen der Mengen ap_K und pp_K werden nicht explizit spezifiziert, sondern sind mit dem semantischen Modell definiert. Zur formalen Definition der Mengen ap_K und pp_K sowie aller hier informell erläuterten Begriffe verweisen wir auf [GS96b], [GS96c] und [GS96a].

Die Semantik gewährleistet, daß sich die Komponenten *vertraulichkeitserhaltend* verhalten: das Verhalten einer Komponente hängt nur von solchen Ports ab, die ihr aktuell zur Verfügung stehen. Aufgrund dieser Eigenschaft ist für mehrere Komponenten das *gemeinsame Verwenden* eines Kanals möglich. Dies bedeutet, daß sie nacheinander ein Zugriffsrecht für den gleichen Kanal erhalten können. Es ist sichergestellt, daß eine Komponente einen gesendeten Port sofort vergißt. Für die Spezifikation mobiler Komponenten bedeutet dies, daß Komponenten Rechte an Kanälen *weiterleiten* können. Damit wird insbesondere gewährleistet, daß aktuell nutzbare Kanalverbindungen umgelenkt werden können.

Eine *mobile Funktion* ist eine benannte, stromverarbeitende Funktion, die sich mobil bzgl. ihrer initialen Vernetzung verhält und die oben beschriebenen Eigenschaften mobiler Funktionen erfüllt. Seien N die Menge aller im System verfügbaren Kanalbezeichner und S_n für $n \in N$ der jeweils zugeordnete Nachrichtentyp. Für eine Funktion f, die sich mobil bzgl. ihrer Mengen I, O und P mit $I \cap O = P \cap (I \cup O) = \emptyset$ verhält, schreiben wir auch

$$f_K \quad \in \quad (N \mapsto \prod_{n \in N} [S_n^*]) \xrightarrow{I,O,P} (N \mapsto \prod_{n \in N} [S_n^*]) \tag{2.9}$$

Auch mobile, dynamische Komponenten werden prädikativ als Mengen stark pulsgetriebener und mobiler stromverarbeitender Funktionen spezifiziert. Das dabei verwendete Prädikat definiert mobile Funktionen, die initial über die Eingabekanäle I empfangen, über die Ausgabekanäle O senden und über private Kanäle aus P verfügen können.

Die auf Seite 22 in Punkt 2. charakterisierten *dynamischen* Systeme können ebenfalls im erweiterten Modell spezifiziert werden. Neue Komponenten werden auf der Basis eines Funktionsaufrufs und mit Hilfe des Kompositionsoperators $\otimes$ in die Systemstruktur eingefügt. Hierbei können einer Komponente K bei ihrer Erzeugung Ports als Parameter übergeben werden, die die initiale Vernetzung von K und damit die Eingliederung von K in das bestehende Netz festlegen. Insbesondere kann hierbei eine private Kanalmenge definiert werden, wodurch K in die Lage versetzt wird, den Aufbau neuer Verbindungen eigenständig zu initiieren. Für die Spezifikation eines dynamischen Systems muß gewährleistet sein, daß die formale Spezifikation der neu erzeugten Komponente existiert. Das Löschen von Komponenten wird dadurch modelliert, daß ihre bestehende aktuelle Schnittstelle vollständig gelöscht wird. Damit kann das Verhalten von K im nächsten Zeitintervall nicht mehr beobachtet werden. Eine Eingliederung in das bestehende System ist ebenfalls nicht mehr möglich, da K keine Ports empfangen kann.

Im folgenden beschreiben wir die Einsatzmöglichkeiten des erweiterten Modells, geben informelle Leitlinien an und legen Spezifikationsschemata fest, die für Spezifikationen mobiler, dynamischer Netze verwendet werden können, siehe hierzu auch [HS96] und [HS97].

Spezifikationsschemata für mobile Systeme

Im folgenden seien A und B mobile Komponenten. Wir beschreiben das Erzeugen neuer und das Umlenken und Löschen bestehender Kanäle. Wir setzen voraus, daß zwischen A und B bereits eine direkte oder indirekte Kommunikationsverbindung besteht[11].

a) Erzeugen eines neuen Kanals

Zwischen A und B soll eine neue Kanalverbindung aufgebaut werden. Dabei unterscheiden wir folgende Fälle:

1. A und B sind aufgrund der aktuell bestehenden Verbindungen bereits direkte Kommunikationspartner. Die Komponente, die aktuell über das Schreibrecht an dem entsprechenden Kanal verfügt, initiiert den Verbindungsaufbau zur Partnerkomponente.

   Falls mehrere Kanalverbindungen zwischen A und B existieren und sowohl A als auch B über Schreibrechte an Verbindungskanälen verfügen, können sowohl A als auch B den Aufbau einer neuen Verbindung initiieren.

2. A und B seien mittels bestehender Verbindungen indirekt über weitere Komponenten durch einen *Pfad* miteinander verbunden. Dieser Pfad stellt sicher, daß Nachrichten von A zu B (bzw. von B zu A) gesendet werden können. Dies ist der Fall, wenn die Rechte an den Kanälen aktuell so vergeben sind, daß A über ein Schreib- und B über ein Leserecht (bzw. umgekehrt) an Kanälen verfügt, die den Pfad definieren. Die Komponente, die den Pfad schreibend nutzt, kann den Verbindungsaufbau initiieren. Zusätzlich muß die Spezifikation der Komponenten, die auf dem Pfad zwischen A und B liegen, gewährleisten, daß der Port, der an A bzw. B zu senden ist, unmittelbar weitergeleitet wird. Über einen derartigen Pfad können Verbindungen auch dann aufgebaut werden, wenn keine direkte Verbindung zwischen A und B besteht.

3. Liegt weder die mit (1) noch die mit (2) beschriebene Situation vor, so können weder A noch B den Verbindungsaufbau initiieren. Sind jedoch beide über eine dritte Komponente C so miteinander verbunden, daß jeweils A und B über das Leserecht an einem Pfad zu Komponente C verfügen, so kann C den Verbindungsaufbau initiieren.

Liegt keiner dieser Fälle vor, so kann zwischen A und B keine Verbindung aufgebaut werden.

Wir geben für die Fälle 1. bis 3. schematisch verwendbare Funktionsgleichungen an. Es gilt: f_A, f_B und f_C seien mobile Funktionen, die jeweils die Spezifikation der Komponenten A, B und C erfüllen, und s sei ein für die Spezifikation typkorrekter Strom.

[11] Da Information nur über bestehenden Verbindungen fließen kann, ist diese Forderung angemessen.

Ohne Beschränkung der Allgemeinheit gehen wir davon aus, daß zwischen A und B eine direkte Verbindung über Kanal $AtoB$ aufgebaut werden soll. Hierbei soll A das Schreib- und B das Leserecht für diese Verbindung erhalten. Für 1. und 2. sei A der Initiator. Für 2. und 3. setzen wir voraus, daß der zwischen A und B existierende Pfad nur über *eine* weitere Komponente C führt und die einzig mögliche Verbindung zwischen A und B darstellt. In 3. initiiert C den Verbindungsaufbau. Wir gehen davon aus, daß das Erzeugen eines neuen Kanals mit dem Empfang einer Nachricht *new* über Kanal in_A in 1. und 2. bzw. über Kanal in_C in 3. ausgelöst wird. Es gelten $pp_A \neq \emptyset$ und $pp_C \neq \emptyset$ als Voraussetzung dafür, daß A und C überhaupt neue Verbindungen erzeugen können.

zu 1: A initiiert den Aufbau einer neuen Verbindung zu B über die bereits bestehende Verbindung über Kanal $AtoB_{old}$. Es gelten: $\{?in_A, !AtoB_{old}\} \subseteq ap_A$ und $?AtoB_{old} \in ap_B$, sowie $?!AtoB \in pp_A$. Die Spezifikation erfolgt mit der Funktionsgleichung:

$$f_A(\{in_A \mapsto \langle new \rangle\} \circ s) \quad = \quad \{AtoB_{old} \mapsto \langle ?AtoB \rangle\} \circ f_A(s) \tag{2.10}$$

Aufgrund der Semantik und der Definition der Mengen *ap* gelten in der nächsten Phase des Systems: $!AtoB \in ap_A$ und $?AtoB \in ap_B$.

zu 2: Zwischen A und B besteht keine direkte Verbindung; es gibt aktuell keinen Port in ap_A, dessen Komplement in ap_B enthalten ist. Es gilt: $\forall p \in ap_A : \; \bar{p} \notin ap_B$.[12]
Die Kanäle $AtoC$ und $CtoB$ bilden den Pfad von A über C zu B. Es gelten : $\{?in_A, !AtoC\} \subseteq ap_A$, $?!AtoB \in pp_A$, $\{?AtoC, !CtoB\} \subseteq ap_C$ und $?CtoB \in ap_B$. Der Verbindungsaufbau wird spezifiziert mit

$$f_A(\{in_A \mapsto \langle new \rangle\} \circ s) \quad = \quad \{AtoC \mapsto \langle ?AtoB \rangle\} \circ f_A(s) \tag{2.11}$$

Damit gilt in der nächsten Phase des Systems $!AtoB \in ap_A$. Die Spezifikation für C muß sicherstellen, daß C den Port $?AtoB$ an B weiterleitet. Dies erfolgt mit:

$$f_C(\{AtoC \mapsto \langle ?AtoB \rangle\} \circ s) \quad = \quad \{CtoB \mapsto \langle ?AtoB \rangle\} \circ f_C(s) \tag{2.12}$$

Sobald B den Port $?AtoB$ empfängt, gilt $?AtoB \in ap_B$.

zu 3: Zwischen A und B existiert weder eine direkte Verbindung noch ein Pfad, den A schreibend nutzen kann. Da C über Kanalverbindungen mit Schreibzugriff sowohl zu A als auch zu B verfügt, kann C den Verbindungsaufbau zwischen A und B initiieren. Sei N die Menge der für das System definierten Kanäle, und es gelte:

$$\begin{aligned}
&\forall ch \in N : \;\; (!ch \in ap_A \Rightarrow ?ch \notin ap_C \cup ap_B) \;\; \wedge \;\; (!ch \in ap_B \Rightarrow ?ch \notin ap_C \cup ap_A) \\
&\wedge \\
&\exists ch_1, ch_2 \in N : \;\; \{!ch_1, !ch_2\} \in ap_C \;\; \wedge \;\; ?ch_1 \in ap_A \;\; \wedge \;\; ?ch_2 \in ap_B
\end{aligned}$$

Für das Schema gelte: $?!AtoB \in pp_C$, $\{?in_C, !CtoA, !CtoB\} \subseteq ap_C$, $?CtoA \in ap_A$ und $?CtoB \in ap_B$. Die Spezifikation des Verbindungsaufbaus erfolgt mit:

$$f_C(\{in_C \mapsto \langle new \rangle\} \circ s) \;\; = \;\; \{CtoA \mapsto \langle !AtoB \rangle, CtoB \mapsto \langle ?AtoB \rangle\} \circ f_C(s) \tag{2.13}$$

[12] $\bar{p}$ bezeichnet das Komplement eines Ports; es gilt beispielsweise $\overline{!AtoB} = ?AtoB$.

Die hier genannten Voraussetzungen gelten nur für den Fall, daß *eine* dritte Komponente C die einzig mögliche Verbindung zwischen A und B darstellt. Falls es weitere Komponenten gibt, die A und B verbinden, müssen die oben angegebenen Voraussetzungen angepaßt werden.

Soll A lesend und B schreibend auf $AtoB$ zugreifen, wird $!AtoB$ an B geschickt.

b) Umlenken einer bestehenden Kanalverbindung

Eine Komponente A kann alle Ports $p \in ap_A \cup pp_A$, die ihr aktuell zur Verfügung stehen, als Nachrichten versenden. Dafür muß A wenigstens einen der Kanäle in ap_A schreibend nutzen können. Durch das Versenden eines Ports p, der zu ap_A gehört, kann A eine *bestehende* Kanalverbindung *umlenken*. Sie gibt das Zugriffsrecht, das sie an diesem Kanal hat, an eine andere Komponente weiter.

Sei in_A ein Kanal, auf dem A aktuell Nachrichten empfangen kann. A sei mit Komponente B über Kanal $AtoB$ verbunden, wobei A über das Schreibrecht verfügt. Es gilt $\{?in_A, !AtoB\} \subseteq ap_A$. Ausgelöst durch den Erhalt der Nachricht $forward$ über Kanal in_A, gibt A das Leserecht an Kanal in_A an B weiter. Die Spezifikation erfolgt gemäß:

$$f_A(\{in_A \mapsto \langle forward \rangle\} \circ s) \quad = \quad \{AtoB \mapsto \langle ?in_A \rangle\} \circ f_A(s) \tag{2.14}$$

c) Löschen eines Kanals

Komponenten können nicht nur Rechte an Kanälen erhalten, sondern diese auch abgeben. (Ein Beispiel war bereits das Weiterleiten von Kanälen.) Das Löschen von Kanälen wird mit der Abgabe von Rechten an Kanälen spezifiziert. Eine Komponente gibt das Zugriffsrecht an dem zu löschenden Kanal an die Komponente zurück, die das komplementäre Recht an diesem Kanal besitzt. Sobald diese Komponente das Zugriffsrecht empfängt, verfügt sie über beide Rechte an dem betroffenen Kanal, der entsprechende Port wird aus der Menge ap gelöscht, und beide Ports werden in die Menge pp aufgenommen.

Die Komponenten A und B seien über den Kanal $AtoB$ verbunden, wobei A über das Schreib- und B über das Leserecht verfügt. N sei die Menge der für das System definierten Kanalbezeichner. Kanal $AtoB$ soll nach Empfang der Nachricht *delete* gelöscht werden, die A über in_A bzw. B über in_B empfangen kann. Es gelten $\{!AtoB, ?in_A\} \subseteq ap_A$ und $\{?AtoB, ?in_B\} \subseteq ap_B$.

1. A initiiert das Löschen von Kanal $AtoB$, indem A den Port $!AtoB$ über die noch bestehende Verbindung $AtoB$ sendet. Hierfür gilt folgendes Schema:

   $$f_A(\{in_A \mapsto \langle delete \rangle\} \circ s) \quad = \quad \{AtoB \mapsto \langle !AtoB \rangle\} \circ f_A(s) \tag{2.15}$$

   Sobald B den Port empfangen hat, gilt $?!AtoB \in pp_B$. $AtoB$ wurde gelöscht.

2. B, als die Komponente, die über das Leserecht an $AtoB$ verfügt, kann das Löschen von $AtoB$ nur dann initiieren, wenn eine der folgenden Voraussetzungen erfüllt ist:

(a) Es gibt eine weitere direkte Verbindung zwischen A und B, die B schreibend nutzen kann. Es gibt somit einen Kanal $ch \in N$ mit $!ch \in ap_B \;\wedge\; ?ch \in ap_A$. Das Löschen von Kanal $AtoB$ erfolgt nach dem Schema:

$$f_B(\{in_B \mapsto \langle\, delete \,\rangle\} \circ s) \;=\; \{ch \mapsto \langle\, ?AtoB \,\rangle\} \circ f_B(s) \qquad (2.16)$$

(b) Es gibt einen Pfad zwischen B und A, den B schreibend nutzen kann. Es gibt also eine dritte Komponente C, für die gilt:

$$\exists ch_1, ch_2 \in N \;:\; !ch_1 \in ap_B \;\wedge\; \{?ch_1, !ch_2\} \subseteq ap_C \;\wedge\; ?ch_2 \in ap_A$$

Für B gilt das Schema

$$f_B(\{in_B \mapsto \langle\, delete \,\rangle\} \circ s) \;=\; \{ch_1 \mapsto \langle\, ?AtoB \,\rangle\} \circ f_B(s) \qquad (2.17)$$

und die Spezifikation für C muß folgende Gleichung enthalten:

$$f_C(\{ch_1 \mapsto \langle\, ?AtoB \,\rangle\} \circ s) \;=\; \{ch_2 \mapsto \langle\, ?AtoB \,\rangle\} \circ f_C(s) \qquad (2.18)$$

Sobald A den Port $?AtoB$ über Kanal ch in Fall (a) und über Kanal ch_2 in Fall (b) empfangen hat, gilt $?!AtoB \in pp_A$.

Existiert aktuell kein Pfad zwischen A und B oder kann B diesen Pfad nicht schreibend nutzen, so ist B nicht in der Lage, den Port $?AtoB$ an A zu senden. Bei der Erstellung einer Spezifikation muß dafür gesorgt werden, daß eine Komponente, die einen Kanal löschen soll, über die dafür benötigten Zugriffsrechte verfügt. Sowohl das Erzeugen und das Weiterleiten als auch das Löschen von Kanälen kann auch für mehrere Kanäle in einem Schritt erfolgen. Die oben für *einen* Kanal beschriebenen Bedingungen gelten dann entsprechend für alle betroffenen Kanäle. Die Ports werden als Sequenz verschickt, und die schematischen Funktionsgleichungen gelten entsprechend, wobei der oben auftretende Port durch die Sequenz ersetzt wird.

Spezifikationsschemata für dynamische Systeme

In *dynamischen* Systemen können Komponenten erzeugt oder aus dem System entfernt werden. Im folgenden geben wir Schemata für beide Möglichkeiten an.

a) Erzeugen einer Komponente

Eine Komponente soll eine Kindkomponente *Child* erzeugen. Die Modellierung erfolgt durch den Aufruf einer Funktion *child*, die das Verhalten von *Child* beschreibt. Im folgenden sei A die erzeugende Komponente. Wir gehen davon aus, daß f_A ein gültiges Verhalten für A beschreibt. Es gelten $ap_A \neq \emptyset$ und $pp_A \neq \emptyset$ sowie $?in_A \in ap_A$. Angestoßen durch die

Nachricht *create* über Kanal in_A erzeugt A eine Kindkomponente. Die Spezifikation erfolgt gemäß dem Muster

$$\begin{aligned}&\exists\ child \in [[Child]]\ :\\ &\quad f_A(\{in_A \mapsto \langle create \rangle\} \circ s)\ =\\ &\qquad (f_A \otimes child(M_{Ports}))(\{ch_1 \mapsto X_1, \ldots, ch_n \mapsto X_n\} \circ s)\end{aligned} \tag{2.19}$$

Dabei gelten:

1. Das Erzeugen der Komponente *Child* wird durch den Aufruf der Funktion *child* modelliert. Mit $\exists\ child \in [[Child]]$ wird die Existenz einer solchen Funktion sichergestellt. Eine Spezifikation für *Child* muß vorliegen.

2. Die neu erzeugte Komponente wird mittels $\otimes$ in die Systemstruktur eingefügt.

3. *Child* erhält die *initialen Vernetzung* durch die Menge M_{Ports}, die sie von A erhält. Es muß $M_{Ports} \subseteq ap_A \cup pp_A$ gelten. Da A dem erzeugten *Child* sowohl aktive als auch private Ports mitgeben kann, wird *Child* in die bestehende Systemstruktur eingebunden *und* erhält die Möglichkeit, eigenständig neue Kanalverbindungen zu erzeugen.

4. Sowohl A als auch *Child* arbeiten auf dem Strom s weiter. Es ist möglich, einer Kindkomponente explizit initiale Nachrichten zu übergeben. Dies geschieht durch das Voranstellen dieser Nachrichten vor s. Im Schema sei $\{?ch_1, \ldots, ?ch_n\} \subseteq M_{Ports}$. Nur *Child* ist in der Lage, die Nachrichten $X_1, \ldots, X_n$ zu empfangen.

Mit diesem Schema können eine oder mehrere Komponenten erzeugt werden. Komponenten, die ohne private Ports erzeugt werden, sind zunächst nicht in der Lage, die Erzeugung neuer Kanäle zu initiieren. Die initiale Schnittstelle einer Kindkomponente sollte niemals leer sein, da damit ihr Verhalten im FOCUS-Modell nicht beobachtbar wäre und sie im Nachhinein nicht mehr in das System eingebunden werden kann.

b) Löschen einer Komponente

Das Löschen einer Komponente wird dadurch modelliert, daß die aktuelle Schnittstelle der Komponente vollständig gelöscht wird. Wir unterscheiden die Fälle:

1. Eine Komponente initiiert das eigene Löschen.

2. Eine Komponente wird durch andere Komponenten gelöscht.

Die Komponente A soll gelöscht werden, und das Löschen wird durch die Nachricht *release* initiiert. N sei die Menge der im System definierten Kanalbezeichner. In unseren Modellierungen kürzen wir eine mobile Funktion f_A, für die aktuell $ap_A = \emptyset$ gilt, mit „`null`" ab. Der Aufruf dieser Funktion tritt dann auf, wenn eine Komponente gelöscht wird.

zu 1: Damit A das Löschen der eigenen Schnittstelle initiieren kann, muß A über mindestens einen Ausgabekanal verfügen. Es gelte $!out_A \in ap_A$. A wird gelöscht, wenn

A alle Zugriffsrechte an Kanälen, die ihre aktuelle Schnittstelle bilden, abgibt. Dies geschieht nach folgendem Schema:

$$f_A(\{in_A \mapsto \langle release \rangle\} \circ s) \quad = \quad \{out_A \mapsto \langle ?in_A, pstr, !out_A \rangle\} \circ \texttt{null} \qquad (2.20)$$

A gibt eine Sequenz bestehend aus allen $p \in ap_A$ über einen ihrer Ausgabekanäle, hier out_A, aus. Es gilt: $\forall p \in ap_A : \ \#(p \copyright (?in_A \circ pstr \circ !out_A)) = 1$.

zu 2: Für A gilt aktuell $\forall ch \in N : \ !ch \notin ap_A$. A kann eigenständig keine Zugriffsrechte abgeben, sich somit selbst nicht löschen. Das Löschen kann nur durch Partnerkomponenten durchgeführt werden, indem diese ihre Zugriffsrechte an Kanälen zu A zurückschicken. Sobald $ap_A = \emptyset$ gilt, ist A gelöscht. Alle Partnerkomponenten müssen eine Gleichung nach einem der Schemata aus Abschnitt c) von Seite 27 enthalten.

Das Mischen der beiden oben genannten Fälle ist möglich: A kann einige Kanäle eigenständig löschen, während andere Kanäle von Partnerkomponenten gelöscht werden.

2.7 Systemstruktur- und Ereignisdiagramme

Zur Erstellung von Spezifikationen können graphische Beschreibungstechniken eingesetzt werden. Wir werden zwei graphische Beschreibungstechniken von FOCUS einsetzen, die *Systemstrukturdiagramme* und zum Einstieg in die Modellierung die *Erweiterten Ereignisdiagramme*. Diese werden im folgenden kurz beschreiben.

Systemstrukturdiagramme (SSDs)

Systemstrukturdiagramme (SSDs) werden dazu eingesetzt, die Struktur eines verteilten Systems und die Schnittstelle einer Basiskomponente graphisch zu erfassen. Komponenten werden durch Kästen, die mit deren Bezeichner versehen sind, und Kanäle durch gerichtete und mit den Bezeichnern und den Nachrichtentypen der Kanäle markierte Kanten dargestellt. Kanten verbinden entweder Komponenten untereinander oder das System mit der Umgebung. Ausgehende Kanten gelten für die Komponente als Ausgabe- und eingehende Kanten als Eingabekanäle. Kanten, für die explizit keine Komponente als Ausgangs- bzw. Zielkomponente angegeben ist, definieren die Verbindung zur Umgebung. SSDs erhalten ihre mathematische Bedeutung durch die direkte Umsetzung in eine syntaktische Darstellung gemäß ANDL. Zur Erhaltung der Übersichtlichkeit verzichten wir auf die Festlegung der Nachrichtentypen und geben nur die Kanal- und Komponentenbezeichner an.

Abbildung 2.7.1 zeigt das SSD für ein verteiltes System `Netz`, das aus drei Subkomponenten A, B und C besteht. Die Komponenten A und B bilden mit ihren Eingabekanälen $InA1$, $InA2$ und InB und die Komponenten B und C mit ihren Ausgabekanälen $OutB$ und $OutC$ die Schnittstelle von `Netz` zur Umgebung.

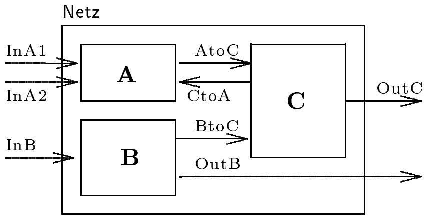


Abbildung 2.7.1: Beispiel für ein SSD

Zur Darstellung des Systemaufbaus eines statischen Systems ist *ein* SSD ausreichend. Dies gilt für mobile, dynamische Systeme im allgemeinen nicht, da sich die Struktur des Systems während eines Systemablaufs mehrfach verändern kann. Wir werden SSDs als graphisches Darstellungsmittel einsetzen, um *Schnappschüsse* aufzuzeigen, in denen die Struktur des Systems unverändert bleibt. Für jede derartige Phase wird ein SSD angegeben. Die graphische Darstellung der Struktur eines mobilen, dynamischen Systems besteht aus einer Folge von SSDs, die alle oder einen Teil der Phasen repräsentieren, die das System durchläuft. Das SSD, welches die initiale Systemkonfiguration darstellt, wird in ANDL umgesetzt.

Erweiterte Ereignisdiagramme (EET's)

Erweiterte Ereignisdiagramme (EETs) werden dazu verwendet, das Zusammenspiel der Komponenten des gesamten Systems bzw. ausgewählter Systemteile durch Beispielabläufe zu veranschaulichen. EETs sind eine Variante der sogenannten *Message Sequence Charts (MSCs)*, die Merkmale der Version MSC'96, siehe [Int96], aufweisen. In einem EET wird das System aus der Sicht der Komponenten beschrieben, und die Systemabläufe werden durch den Austausch von Nachrichten zwischen den Komponenten dargestellt.

Ein elementares EET besteht aus vertikalen Achsen und horizontalen, gerichteten Kanten. Die Achsen repräsentieren die Komponenten und sind mit deren Bezeichnern markiert. Die Kanten (Pfeile) verbinden Achsen miteinander und veranschaulichen die Wechselwirkungen zwischen Komponenten. Sie repräsentieren Kommunikationsereignisse und werden mit den Bezeichnern der ausgetauschten Nachrichten markiert. Entsprechend zu den SSDs repräsentieren Pfeile, denen explizit keine Ausgangs- bzw. Zielachse zugeordnet ist, Interaktionen mit der Umgebung. Innerhalb eines elementaren EETs werden zeitliche Abhängigkeiten der Nachrichten dokumentiert, die zwischen den Komponenten ausgetauscht werden. Die Pfeile entlang einer Achse sind zeitlich geordnet: durch einen Pfeil, markiert mit a, der oberhalb eines Pfeiles markiert mit b angeordnet ist, wird veranschaulicht, daß die Nachricht a zeitlich vor b auftritt. In Kapitel 4 werden Beispielabläufe durch EETs veranschaulicht.

Erweiterungen elementarer EETs ermöglichen hierarchische Sichten auf Systemabläufe:

- EETs können gruppiert werden, um *alternative* Systemabläufe zusammenzufassen. Eine Gruppe wird mit einem Bezeichner versehen. Auf die Bezeichner kann in einer Box Bezug genommen werden.

- Abschnitte eines EETs können von einer alle Achsen umfassenden *Box* überdeckt werden. Boxen erhalten einen Bezeichner, der sich auf eine Gruppe von EETs bezieht.

In dieser Darstellungsform werden EETs von der Beschreibung exemplarischer Abläufe verwendet. Für weitere Erläuterungen zu EET's siehe [BHS96] und [HSE97]. Wir zeigen hier kein Beispiel für ein EET und verweisen auf die Abschnitte 4.3.2 und 4.4.5.

2.8 Die Kernsprache ANDL

Für die Beschreibung von Komponenten und Systemen auf textueller Ebene steht in FOCUS die Kernsprache ANDL (*Agent Network Description Language*) zur Verfügung. In programmiersprachlicher Notation bietet ANDL einfache Konstrukte zur Beschreibung der Schnittstelle und der Systemstruktur. Mit einer ANDL-Spezifikation wird die Semantik einer Spezifikation, und dabei insbesondere das gewählte semantische Modell festgelegt. Eine ANDL-Spezifikation ersetzt die explizite Festlegung des Prädikats und die Typisierung der Funktionen. Bei allen im folgenden gezeigten Spezifikationen wird die Semantik unter Verwendung von ANDL und das Verhalten mittels Funktionsgleichungen mit den festgelegten Schemata erstellt. Im folgenden beschreiben wir die syntaktischen Festlegungen von ANDL informell und beispielhaft. Für detaillierte Erklärungen verweisen wir auf [SS95] und für Erweiterungen zur Spezifikation zeitabhängiger Komponenten auf [Hin96] und [Hin97].

Basiskomponenten

Die Beschreibung einer Basiskomponente umfaßt die Festlegung ihrer Schnittstelle und die Spezifikation ihres Verhaltens. Diese Zweiteilung wird durch die entsprechende Aufspaltung in der ANDL-Spezifikation unterstützt. Im Rumpf werden ein Platzhalter und ein Verweis für die Verhaltensbeschreibung angegeben, die im konstruktiven Spezifikationsstil mittels der Schemata gesondert erstellt wird. Im Kopf wird die Schnittstelle durch die Festlegung eines Bezeichners für die Komponente, die Beschreibungen der Ein- und Ausgabekanäle sowie (für mobile Komponenten) der privaten Kanäle definiert. Eine Kanalbeschreibung besteht aus einem Bezeichner und dem zugehörigen Nachrichtentyp. Die Schnittstellenbeschreibung einer Komponente kann aus einem zugehörigen SSD entnommen werden.

Die semantische Variante der Spezifikation wird mittels spezieller Schlüsselwörter festgelegt. `time dependent` bzw. `time independent` legt fest, ob die Komponente im zeitunabhängigen oder zeitabhängigen Format spezifiziert wird. Die Schlüsselwörter `wp` bzw. `sp` zeichnen Spezifikationen aus, deren Funktionen schwach (weak) bzw. stark pulsgetrieben sind. Das Schlüsselwort `mobile` charakterisiert Komponenten, die als mobile, dynamische Komponenten spezifiziert werden. Alle in der vorliegenden Arbeit erstellten Spezifikationen basieren auf stark pulsgetriebenen, mobilen Funktionen im zeitabhängigen Format mit point-to-point-Kommunikation. Wir verzichten auf die explizite Nennung der Schlüsselwörter.

Als Beispiel zeigen wir in Abbildung 2.8.1 die ANDL-Spezifikation der Basiskomponente A aus Abbildung 2.7.1 von Seite 31. A wird als mobile Komponente mit einer leeren Menge privater Kanäle spezifiziert. S_{InA1}, S_{InA2} und S_{AtoB} seien die für die entsprechenden Kanäle festgelegten Nachrichtentypen.

agent A	
input channels	$InA1 : S_{InA1},\ InA2 : S_{InA2},\ CtoA : S_{CtoA}$
output channels	$AtoB : S_{AtoB}$
private channels	$\emptyset$
is basic	
$\boldsymbol{f_A}$	mit der Spezifikation von Seite pp
end A	

Abbildung 2.8.1: ANDL-Spezifikation von A

Diese ANDL-Spezifikation enthält sämtliche Informationen, die zur Festlegung der Semantik der Spezifikation von A benötigt wird. Die in Abbildung 2.8.1 gezeigte Spezifikation ersetzt

$$[\![\mathtt{A}]\!] \ = \ \{\ f_A\ :\ Type_A \mid P_A.f_A\ \wedge\ f_A \text{ stark pulsgetrieben und mobil } \}$$

Hierbei kürzen wir mit $Type_A$ den Funktionstyp für mobile stromverarbeitende Funktionen ab, der sich mit (2.9) von Seite 24 ergibt. Die ANDL-Spezifikation enthält in Kombination mit den Festlegungen der Nachrichtentypen alle für die Festlegung des Funktionstyps benötigten Informationen. Für mobile Komponenten wird die initiale Schnittstelle vorgegeben, so daß sich die Mengen I, O und P, siehe Abschnitt 2.6, ebenfalls bestimmen lassen. Das Prädikat $P_A\ :\ Type_A \to \mathbb{B}$ legt das Verhalten der Komponente A fest und charakterisiert alle mobilen Funktionen, die auf der Seite „pp“ mittels konjugierter Funktionsgleichungen angegeben werden. Eine derartige Spezifikation wird textuell erarbeitet. Die Formalisierung ergibt sich aus deren schematischer Umsetzung und wird in folgendem Rahmen gezeigt:

Funktionsgleichungen für f_A
Quantifizierung der verwendeten Variablen
Funktionsgleichungen gemäß den Schemata
wobei: *für alle Gleichungen gültige Abkürzungen*

Verteilte Systeme

Entsprechend zu den Basiskomponenten besteht die Beschreibung eines Netzes aus zwei Teilen. Der Kopf der Spezifikation enthält die Beschreibung der Schnittstelle des verteilten Systems. Im Rumpf wird die Vernetzungsstruktur des Systems angegeben, dessen Verhalten sich aus dem Verhalten der einzelnen Komponenten ableiten läßt. Eine Beschreibung der Netzstruktur besteht aus einer Menge von Gleichungen der Form

$$\ll \text{ Ausgabekanäle } \gg \quad = \quad \text{Komponente } \ll \text{ Eingabekanäle } \gg$$

Somit werden verschiedene Verbindungsstrukturen auf einheitliche Art und Weise dargestellt. Die in der Schnittstellenbeschreibung angegebenen Kanäle bilden die Verbindung des Systems zu seiner Umgebung. In den Gleichungen der oben genannten Form treten auch systeminterne Kanalverbindungen auf, die die Verbindungen der Komponenten untereinander festlegen. Für verteilte Systeme, die als mobile, dynamische Systeme spezifiziert werden, wird die initiale Vernetzung des Systems angegeben.

Das in Abbildung 2.7.1 gezeigte Beispielsystem wird durch folgende ANDL-Spezifikation beschrieben. Das Verhalten des Systems `Netz` ergibt sich durch Komposition der Basiskomponenten A, B und C, deren Verhalten durch Funktionsgleichungen festgelegt wird.

$$
\begin{array}{ll}
\textsf{agent } \texttt{Netz} & \\
\quad \textsf{input channels} & InA1 \ : \ S_{InA1}, InA2 \ : \ S_{InA_2}, InB \ : \ S_{InB} \\
\quad \textsf{output channels} & OutC \ : \ S_{OutC} \\
\textsf{is network} & \\
\quad \ll AtoB \gg & = \ \texttt{A} \ll InA1, InA2, CtoA \gg \ ; \\
\quad \ll BtoC, OutB \gg & = \ \texttt{B} \ll InB \gg \ ; \\
\quad \ll CtoB, OutC \gg & = \ \texttt{C} \ll AtoC, BtoC \gg \\
\textsf{end } \texttt{Netz} &
\end{array}
$$

Abbildung 2.8.2: ANDL-Spezifikation des verteilten Systems `Netz`

Kapitel 3

Betriebssysteme

In diesem Kapitel geben wir einen Überblick über die zentralen Aufgaben eines Betriebssystems und gehen dabei auf die wesentliche Betriebssystemliteratur ein. Zum besseren Verständnis aller in den folgenden Kapiteln erstellten Modellierungen wird die von uns eingenommene Sichtweise der Funktionalität eines Betriebssystems erläutert und festgelegt. Wir geben keine grundlegende und umfassende Einführung in Betriebssysteme und halten dieses Kapitel möglichst kurz, da viele Erläuterungen und Festlegungen entweder in späteren Kapiteln aufgegriffen werden oder in der genannten Literatur zu finden sind.

3.1 Was ist ein Betriebssystem?

Betriebssysteme sind eines der klassischen Anwendungsgebiete aus der praktischen, technischen und systemnahen Informatik. Ein *Betriebssystem* umfaßt alle Programme, Daten und Dienste eines Rechensystems, die, ohne auf eine bestimmte Anwendung zugeschnitten zu sein, die Nutzung eines Rechensystems ermöglichen. Ein *Rechensystem* ist ein technisches System, das in der Lage ist, Informationen zu speichern und zu verarbeiten, so daß es vom Benutzer zur Durchführung der geforderten Aufgaben eingesetzt werden kann. Ein Rechensystem ist geprägt durch die Hard- und Software, aus der es besteht, wobei insbesondere das Betriebssystem zur Software gehört.

Die Hardware eines Rechensystems ist mit festgelegten Eigenschaften vorgegeben. Sie bildet die physikalische Realität und wird durch entsprechende Software geeignet nutzbar gemacht. Für Benutzer wird die Hardwarekonfiguration so bereitgestellt, daß Anwendungen durchgeführt werden können, ohne daß technische Details bekannt sein oder Benutzer in Maschinensprache programmieren müssen. Das Betriebssystem stellt die *Schnittstelle* zwischen Hardware und Benutzer bereit. In einem Rechensystem stehen Ressourcen (Betriebsmittel), wie beispielsweise *Speicher* oder *Prozessoren*, meist in nicht ausreichender Anzahl zur Verfügung. Zur Durchführung von Anwendungen werden diese Ressourcen benötigt. Das Betriebssystem sorgt für eine sinnvolle und faire Nutzung dieser Ressourcen und übernimmt somit vor allem eine Vielzahl verschiedenster *Verwaltungsaufgaben*, die einem Anwender die zuverlässige Ausführung seiner Berechnungen zusichern, aber für ihn nicht direkt sichtbar sind.

Für eine systematische Behandlung der Betriebssystemproblematik sind drei unterschiedliche Sichten zu beachten. Die Schnittstellenfunktion eines Betriebssystems ist dadurch charakterisiert, daß es den Benutzern den Zugang zum Rechensystem ermöglicht und Aufträge zur Ausführung von Berechnungen entgegennimmt. Hierfür müssen geeignete Systemkomponenten zur Verfügung gestellt werden. Diese Aspekte sind vor allem für die *Benutzer* eines Rechensystems von Interesse. Die zentrale Aufgabe eines Betriebssystems, charakterisiert durch den *Verwaltungsaspekt*, umfaßt die

Planung, *Organisation* und *Kontrolle* aller Berechnungen,

die zur Ausführung der erteilten Aufträge erforderlich sind. Insgesamt übernimmt das Betriebssystem eine Brückenfunktion zwischen dem (hohen) Abstraktionsniveau der Benutzer und dem (niedrigen) Abstraktionsniveau der Hardware. Diese Brücke muß geeignet definiert sein und systematisch konstruiert werden. Diese Forderung entspricht den Aufgaben, die im weitesten Sinne in den Bereich des *Software-Engineering* eingeordnet werden können: Durch eine systematische Vorgehensweise sollte möglichst unter Zuhilfenahme formaler Methoden ein beherrschbares, strukturiertes Betriebssystem konstruiert werden. Durch die formale Modellierung der Konzepte, die wir erstellen werden, wird ein erster Schritt in diese Richtung getan. Eine Beschreibung der oben genannten drei *Sichten* auf ein Betriebssystem ist in [Spi95] und auch in [Spi97a] zu finden.

Das Betriebssystem stellt das *Management* eines Rechensystems. Die Realisierung der Managementaufgaben erfolgt durch verschiedene Verwaltungsbereiche, wie die

Prozessorverwaltung zur Ausnutzung der Rechenleistung des Prozessors und zur Durchführung von Berechnungen;

Speicherverwaltung zur Erfüllung des Speicherbedarfs mit Hintergrund- und Arbeitsspeicher;

Prozeßkooperation zur Bereitstellung der Möglichkeit, Berechnungen in Kooperation durchzuführen;

Prozeßverwaltung zur Erzeugung und Auflösung von Prozessen, wobei Ressourcen entsprechend zur Verfügung gestellt bzw. entgegengenommen werden.

Zum Betriebssystem gehören weiter die Geräteverwaltung und der Betriebssystemkern sowie Dateien als Speichermedium für Benutzer, worauf wir im folgenden nicht eingehen werden. Insgesamt besteht eine der zentralen Aufgaben darin, die Prozesse, also die Ausführung der Berechnungen, zu *koordinieren* und hierfür Konzepte und Verfahren bereitzustellen.

Das Management umfaßt das *Buchführen* über freie und belegte Ressourcen, die Zuteilung freier und die Zurücknahme frei gewordener Ressourcen. Zur Gewährleistung der Managementaufgaben muß das Betriebssystem Konzepte bereitstellen, *wie* die Ressourcen *geeignet* zu verwalten sind. Die Entscheidung darüber, welche Kriterien die Eigenschaft *geeignet* erfüllen muß, ist von einer Vielzahl von Faktoren abhängig und kann bzgl. der eingesetzten Verfahren nicht immer eindeutig beantwortet werden. Vergleiche hierzu beispielsweise die verschiedenen Strategien für *Scheduling* oder zur *Auslagerung* von Seiten bei virtuellem Speicher und deren Klassifizierung und Bewertung in den Kapiteln 2.4 und 3.4 in [Tan94], 5.3 und 9.5 in [GS94] sowie 5.2 und 6.4.3 in [Spi95].

Betriebssysteme sind durch zwei wesentliche und unterschiedliche Aspekte charakterisiert:

1. Die Definition *abstrakter* (oder virtueller) *Maschinen*, die die Hardware durch entsprechende Software direkt und komfortabel nutzbar macht.

2. Die *Ressourcenverwaltung* und alle dazu gehörenden Strategien und Verfahren, die nötig sind, um die Ressourcen fair und geeignet zur Verfügung zu stellen.

Diese Zweiteilung wird auch in der klassischen Literatur, siehe [MOO87], [PS85], [GS94], [TB93], [Tan90] und [Tan92] sowie [Sta92], deutlich: durch die direkte Benennung beider Aspekte, wie beispielsweise in Kapitel 1.1 von [Tan94], die Beschreibung verschiedener Sichten in [Fin88], in dem ein Betriebssystem explizit unter dem Gesichtspunkt „*Ressource Principle*“ betrachtet wird, oder indirekt durch die Strukturierung der Literatur durch ihre Kapitel. Beschreibungen realisierter Betriebssysteme bestehen weitgehend aus der Darstellung der abstrakten Maschine; vergleiche beispielsweise Unix in [GO95].

Wir beschränken uns auf den Managementaspekt, da uns dieser im Hinblick auf die Möglichkeiten zur Strukturierung eines Systems und zur Beschreibung der charakteristischen Merkmale eines Betriebssystems mit formalen Methoden geeignet erscheint. Bei der Strukturierung des von uns entwickelten verteilten Systems sind folgende Aspekte zu berücksichtigen: Das Betriebssystem muß die zur Verfügung stehenden Ressourcen nutzen, wodurch sich auf der Basis der zugrundeliegenden Hardwarekomponenten eine entsprechende Strukturierung des Systems ergibt. Neben dieser räumlich bedingten Strukturierung sind weitere Aufteilungen zu charakterisieren, die sich aus der geforderten Funktionalität und der daraus resultierenden Aufgabenteilung ergeben. Wie sich dies bei der formalen Modellierung eines Systems niederschlägt, wird in den folgenden Kapiteln deutlich. Wir werden Focus dafür einsetzen, um eine erste formale Modellierung eines verteilten Systems, das die wesentlichen Charakteristika eines Betriebssystems aufweist, systematisch zu erstellen.

3.2 Betriebssystemprozesse

In Abschnitt 3.1 wurde erklärt, daß ein Betriebssystem die Software darstellt, die nötig ist, um die Aufträge eines Benutzers vor allem möglichst komfortabel ausführen zu können. Die Aufträge, die ein Benutzer an das System stellt, sind üblicherweise Berechnungen, die er unter Ausnutzung der Rechenleistung eines Rechensystems durchführen möchte. Das fundamentale Konzept für den Umgang mit dieser Aufgabenstellung bilden die Prozesse in Betriebssystemen, siehe auch [GS94] und [Tan92] und die weitere Standardliteratur.

Prozesse stehen für die Aufträge, die an das System gestellt werden; sie repräsentieren die Benutzer und bilden die Schnittstelle zu diesen. Wir gehen davon aus, daß alle Berechnungen als Prozesse ausgeführt werden. Insbesondere werden auch solche Berechnungen, die sich bei der Ausführung von Verwaltungsaufgaben ergeben, im System als Prozesse, sogenannte Systemprozesse, behandelt. Wir betrachten im folgenden nur Benutzerprozesse.

Prozesse sind die abstrakten, aktiven Komponenten innerhalb eines Rechensystems. Ein Prozeß ist ein Strukturelement, das es erlaubt, die Beschreibung einer Berechnung (also das *Programm*) von deren Ausführung zu trennen. Die einen Prozeß beschreibende Information befindet sich üblicherweise in einem *Kontrollblock*, vergleiche [Tan92] oder [GS94]. Prozesse werden auch als *abstrakte Prozessoren* verstanden, die festgelegte Berechnungen ausführen, wobei das zugehörige Programm die durchzuführenden Schritte festlegt. Die Ausführung von Berechnungen erfolgt ausschließlich mit dem realen Prozessor der Hardwarekonfiguration, siehe auch Abschnitt 3.3. Vor allem sind Prozesse das Konzept, an dem die zur Ausführung einer Berechnung von einem Betriebssystem durchzusetzenden Verwaltungsaufgaben demonstriert werden.

Die Funktionsweise eines Betriebssystems wird in erster Linie anhand eine Menge von Prozessen demonstriert, von denen einige zueinander in Verbindung stehen, während andere unabhängig voneinander ausgeführt werden können. Ein- und Mehrprozessorsysteme bestimmen die Anzahl der Prozesse, die höchstens gleichzeitig aktiv sein können. Ein Betriebssystemprozeß steht für ein Programm, das gestartet und dessen Ausführung noch nicht beendet ist. Im allgemeinen benötigt ein Prozeß Ressourcen, wie Eingabedaten, Speicher und insbesondere den Prozessor. Das so repräsentierte Programm muß nicht notwendigerweise in Ausführung sein, sondern kann auf die Zuteilung von Ressourcen warten, die es für die Fortführung der Berechnung benötigt. Die *Phasen*, die ein Prozeß durchläuft, werden durch ein Zustandsübergangsdiagramm dargestellt, siehe Abbildung 3.2.1. Dieses Diagramm wird üblicherweise zur Beschreibung der Funktionsweise eines Prozesses herangezogen; vergleiche die Abschnitte 4.1 in [PS85], 2.1.1 in [Tan94], 5.1.2 in [Spi95] und auch 3.1 in [GS94]. Die Zustände und Zustandsübergänge charakterisieren die Managementaufgaben, die ein Betriebssystem erfüllen muß, um eine Berechnung bzw. einen Prozeß auszuführen. Wir gehen davon aus, daß Aufträge zur Ausführung einer Berechnung an das System erteilt werden. Dies geschieht durch Programme, für die, falls sie von einem berechtigten Benutzer stammen, ein entsprechender Prozeß erzeugt wird.

Wir erklären die Zustandsübergänge und konzentrieren uns dabei auf die Verwaltungsaufgaben; die Numerierung entspricht den Markierungen an den Kanten in Abbildung 3.2.1.

1) Um einen Prozeß erzeugen zu können, wird Speicherplatz im Hintergrundspeicher und insbesondere im Arbeitsspeicher benötigt. Steht genügend Speicherplatz zur Verfügung, wird der Prozeß erzeugt, und die Berechnung kann ausgeführt werden.

2) Die Berechnung eines Prozesses, dem bis auf den Prozessor alle benötigten Ressourcen zur Verfügung stehen, kann nur mit dem Prozessor fortschreiten. Sobald ihm dieser zugeteilt wird, kann die Berechnung ausgeführt werden.

3) Der Prozeß verfügt über den Prozessor, und die Ausführung der Berechnung schreitet voran, da alle benötigten Ressourcen zur Verfügung stehen. Nach einer festgelegten Zeitspanne wird ihm der Prozessor entzogen.

4) Der Prozeß verfügt über den Prozessor. Zur Ausführung der Berechnung benötigt er weitere Ressourcen, wie Speicher oder die Kooperation mit einem anderen Prozeß. Da diese Ressourcen nicht zur Verfügung stehen, gibt der Prozeß den Prozessor frei.

5) Der Prozeß wartet auf die Zuteilung benötigter Ressourcen. Sobald ihm diese zugeteilt werden, benötigt er den Prozessor zur Fortsetzung der Berechnung.

6) Mit dem Abschluß der Berechnung terminiert der Prozeß und wird aufgelöst.

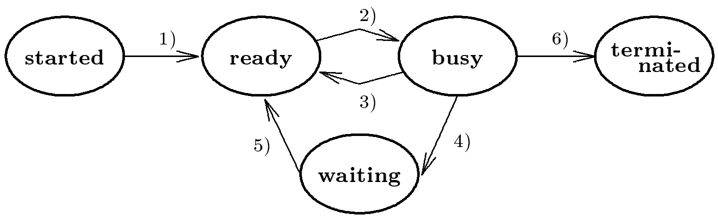


Abbildung 3.2.1: Prozeßzustände

Die in Abbildung 3.2.1 dargestellten Zustandsübergänge resultieren aus den Managementaufgaben eines Betriebssystems. Die Zustände veranschaulichen die Phasen, die Prozesse bzw. auszuführende Berechnungen durchlaufen, und werden bei der Modellierung der Prozesse in Verbindung mit den unterschiedlichen Verwaltungsaspekten eine wesentliche Rolle spielen. Die Zustände sind für den Benutzer eines Rechensystems nicht sichtbar und sind nur für die Modellierung des Ressourcenmanagements relevant. Aus Anwendersicht sind explizite Datenzustände notwendig, um die Berechnungen, die der Prozeß durchführt, zu speichern und konkrete Rechenergebnisse verwenden zu können. Dies wird durch die Speicher eines Rechensystems gewährleistet.

Aus den Zuständen und den Zustandsübergängen leiten wir die wesentlichen Managementaufgaben eines Betriebssystems ab. Ohne den Prozessor als zentrale Komponente des Systems, kann die Ausführung einer Berechnung nicht fortschreiten, dies wird durch die *Prozessorverwaltung* abgedeckt, siehe Kapitel 4. Der Zustand *waiting* wird in zwei Wartezustände aufgeteilt: Zum einen wird Speicherplatz im Arbeitsspeicher benötigt, auf den der Prozessor zugreifen kann; dies wird durch die *Speicherverwaltung* abgedeckt, siehe Kapitel 5. Zum anderen muß ein Prozeß möglicherweise mit anderen Prozessen kooperieren und diese sind zu einer Kooperation noch nicht bereit. Diesen Bereich decken wir durch die Behandlung *nachrichtenorientierter Systeme* ab, siehe Kapitel 6. Zur abschließenden Integration der drei genannten Bereiche sowie für die Erzeugung und Terminierung von Prozessen behandeln wir Konzepte der *Prozeßverwaltung* in Kapitel 7.

3.3 Managementaufgaben eines Betriebssystems

In Abschnitt 3.1 haben wir bereits erklärt, daß wir ein Betriebssystem als den *Verwalter* und *Manager* aller Ressourcen und Aufgaben, die zur Durchführung von Berechnungen erforderlich sind, betrachten. In diesem Abschnitt gehen wir kurz auf die einzelnen, von uns später behandelten Bereiche ein und beschreiben die wesentlichen Aufgabenstellungen. Deren Charakteristika werden in den Kapiteln 4 bis 7 detaillierter und zugeschnitten auf die Anforderungen an die formale Modellierung beschrieben.

Die Aufgabe der Ressourcenverwaltung besteht darin, die im Rechensystem meist in nicht ausreichender Anzahl vorhandenen Betriebsmittel nach festgelegten Maßstäben zu verwalten. Die Ressourcen müssen *fair* verwaltet werden, d.h. jede Komponente, die ein Betriebsmittel benötigt, erhält dieses auch irgendwann. In Abschnitt 3.2 wurde beschrieben, daß Prozesse das wesentliche Konzept in Betriebssystemen sind. Sie spielen bei der Behandlung der Managementaufgaben eine zentrale Rolle. Ausgehend von den Zuständen, die zur Beschreibung eines Prozesses eingesetzt werden, ergeben sich folgende Anforderungen an die Verwaltung der Betriebsmittel:

1. Zur Ausführung einer Berechnung wird der Prozessor benötigt. Es muß dafür gesorgt werden, daß Prozesse den Prozessor zugeteilt bekommen. Sie dürfen diesen nur für eine *angemessene* Zeitspanne belegen, damit er anderen Prozessen zugeteilt werden kann. Die hierfür erforderlichen Maßnahmen umfaßt die *Prozessorverwaltung*.

2. Prozesse benötigen Speicherplatz, und der Prozessor muß auf Speicherplatz im Arbeitsspeicher, der dem Prozeß zugeordnet ist, zugreifen können. Da der Arbeitsspeicher nur begrenzte Kapazität hat, sind hier Maßnahmen zu ergreifen, damit ein Prozeß bei Bedarf über den Arbeitsspeicher verfügen kann. Dieser Bereich wird mit der *Speicherverwaltung* abgedeckt.

3. Da Berechnungen auch von mehreren Prozessen gemeinsam durchgeführt werden können, müssen hierfür geeignete Verfahren bereitgestellt werden. Insbesondere kann nicht notwendigerweise jeder Prozeß mit jedem anderen kooperieren. Die hierfür benötigten Verfahren werden durch die *Prozeßkooperation* gewährleistet.

4. Nur *berechtigte* Benutzer dürfen ein System benutzen. Ein Prozeß zur Ausführung eines Benutzerauftrags kann nur erzeugt werden, wenn für ihn genügend Speicherplatz vorhanden ist. Neue Prozesse werden in das bestehende System so integriert, daß sie bei der Verwaltung der Betriebsmittel berücksichtigt werden. Hierfür sind koordinierende Maßnahmen zu ergreifen, die die *Prozeßverwaltung* umfaßt.

3.3.1 Prozessorverwaltung

Die Realisierung der Rechenfähigkeit eines Rechensystems erfolgt mit den verfügbaren Prozessoren. Da es im allgemeinen weniger Prozessoren als rechenbereite Prozesse gibt, muß das

Betriebsmittel *Prozessor* geeignet verwaltet werden. Diese Aufgabe übernimmt die Prozessorverwaltung. Die Betriebssystemaufgabe besteht darin, darüber zu entscheiden, welcher Prozeß als nächstes einen Prozessor zugeteilt bekommt. Der Teil des Betriebssystems, der hierfür zuständig ist, ist der *Scheduler* mit einem Scheduling-Algorithmus.

Der Scheduler hat die Aufgabe, auf der Basis einer vorgegebenen Strategie Entscheidungen zu treffen. Kriterien zur Auswahl eines Schedulingalgorithmus sind beispielsweise

1. *Fairneß:* Jeder Prozeß erhält den Prozessor nach endlicher Zeit.
2. *Effizienz:* Der Prozessor ist immer ausgelastet.
3. *Durchsatz:* Die Anzahl der Prozesse, die in einer festgelegten Zeitspanne ausgeführt werden, wird maximiert.

Es gibt viele spezielle Scheduling-Algorithmen, deren Schwerpunkte entweder auf ein Kriterium ausgerichtet sind oder einen Kompromiß zwischen mehreren Kriterien bilden. Die Vor- und Nachteile dieser Algorithmen werden ausführlich in der Literatur behandelt, siehe die Kapitel 2.4 in [Tan94], 4.3 in [PS85] oder 6.2 in [Sta92]. Ein weit verbreiteter Algorithmus heißt *Round-Robin*: Jeder Prozeß erhält ein bestimmtes Zeitintervall, ein *Quantum*, das für seine Ausführung, also die Kopplung an den Prozessor, gültig ist. Sobald das Quantum abgelaufen ist, wird der Prozeß suspendiert. Der Prozessor wird ihm entzogen und an einen anderen Prozeß gegeben.

Für das Umschalten zum nächsten ausführbereiten Prozeß und die Einhaltung des Quantums ist der *Dispatcher* zuständig. Wir spezifizieren in Kapitel 4 ein System, in dem die Verwaltung der ausführbereiten Prozesse, das Suspendieren von Prozessen und die Weitergabe des Prozessors gemäß Round-Robin enthalten sind. Auf die Modellierung unterschiedlicher Schedulingalgorithmen gehen wir nicht ein. Zusätzlich wird ein System mit mehreren Prozessoren angeben, wobei alternativ eine zentrale sowie eine dezentrale Verwaltung der Prozessoren modelliert werden.

3.3.2 Speicherverwaltung

Zum Rechensystem gehören der Haupt- oder *Arbeitsspeicher* (AS), auf den der Prozessor direkt zugreifen kann, und der Platten- oder *Hintergrundspeicher* (HS). Für einen Prozeß muß festgelegt werden, aus welchen Einzelschritten die Berechnung besteht; dies wird durch das Programm gewährleistet. Zusätzlich benötigt ein Prozeß Speicherplatz, um das Berechnungsergebnis sowie Teilergebnisse zu speichern. Während der Ausführung der Berechnung wird Speicherplatz im Arbeitsspeicher benötigt. Da der Arbeitsspeicher üblicherweise nicht groß genug ist, um alle Anforderungen zu erfüllen, muß es möglich sein, Prozesse sowohl im Hintergrund- als auch im Arbeitsspeicher zu halten.

Die Aufgabe der Speicherverwaltung eines Betriebssystems besteht darin, die freien und belegten Speicherbereiche von Arbeits- *und* Hintergrundspeicher zu verwalten, Speicherbereiche an Prozesse zuzuweisen und frei werdende Speicherplätze entgegenzunehmen. Die sogenannten *Auslagerungen*, also die Bereitstellung geforderter Daten im Arbeitsspeicher und die Sicherung aktuell nicht benötigter Daten im Hintergrundspeicher müssen durchgeführt werden. Speicherverwaltungen werden danach unterschieden, ob Prozesse während ihrer Ausführung zwischen AS und HS transferieren (*Swapping* und *Paging*), oder nicht. Die Speicherverwaltung hat weiter die Aufgabe, freie Speicherbereiche zu ausreichend großen Bereichen zusammenzufassen. Hierzu gibt es verschiedene Verfahren, die beispielsweise in den Kapiteln 3.2 in [Tan94] oder 5 in [Fin88] beschrieben sind.

Damit ein Benutzer das Programm nicht selbst in kleine, modulare Teile, die eine festgelegte Größe haben, aufteilen muß, wurde das Konzept des *virtuellen* Speichers entwickelt, siehe die Kapitel 9 in [GS94] oder 3.3 in [Tan94]. Die Idee besteht darin, daß der von einem Prozeß benötigte Speicherplatz die Größe des für den Prozeß zugänglichen Speicherplatzes überschreiten darf. Das Betriebssystem sorgt dafür, daß die aktuell benötigten Teile im Arbeitsspeicher zur Verfügung stehen. Die Entscheidungen darüber, welche Teile aus dem Arbeitsspeicher ausgelagert werden können und sollen, werden durch sogenannte *Seitenersetzungsalgorithmen* getroffen. Diese werden mittels verschiedener Kriterien bewertet; siehe beispielsweise die Kapitel 3.4 in [Tan94], 6.4.2 in [Spi95] und 9.5 in [GS94]. Wir werden in Kapitel 5 eine Lösung mit virtuellem Speicher und einer Ersetzungsstrategie gemäß LRU (*Least Recently Used*) modellieren.

3.3.3 Prozeßkooperation

Mit der Prozessorverwaltung wird dafür gesorgt, daß alle Berechnungen der Prozesse fortschreiten können. Daneben sollten auch Konzepte dafür angeboten werden, Prozesse für kooperative Problemlösungen einsetzen zu können. Dabei sollen Prozesse durch ihre Berechnungen Beiträge zu gemeinsamen Lösungen mit anderen Prozessen leisten. Um dieses Ziel zu erreichen, sind Abstimmungen der Prozesse untereinander notwendig. Eine einfache Lösung zur Abstimmung von Prozessen untereinander besteht darin, daß ein Prozeß das Ergebnis, das ein anderer Prozeß bei seiner Terminierung liefert, verwendet, vergleiche beispielsweise Kapitel 9.2 in [Fin88]. Die verschiedenen Lösungen von der Nutzung *gemeinsamer Speicher* bis hin zu *nachrichtenorientierten* Systemen sind in der Literatur beschrieben, siehe [PS85] für die Beschreibung der Lösungen mit gemeinsamem Speicher sowie [Tan94] und [Spi95] zur Beschreibung der Lösungen mit Nachrichtenaustausch.

Da Focus auf Nachrichtenaustausch ausgerichtet ist, werden wir eine Prozeßkooperation modellieren, in der Nachrichtentransporte (*Message Passing*) mit den beiden Basisoperationen `send` und `receive` möglich sind. Dabei werden *blockierendes Senden* und *nichtblockierendes* Senden unterschieden. Diese Konzepte werden nur in wenigen Systemen alternativ angeboten; die Entscheidung für eine der Varianten wird üblicherweise vom Systementwickler getroffen. In Kapitel 6 werden die Funktionsweisen beider Alternativen erläutert und beide Varianten modelliert. Für weitere Erläuterungen wird vor allem auf die Kapitel 2.2.8 in [Tan94] und 2.1 in [Spi95] verwiesen.

3.3.4 Prozeßverwaltung

Die Aufträge, die der Benutzer an ein Rechensystem gibt, sind Berechnungen, die unter Zuhilfenahme der gegebenen Rechenleistung ausgeführt werden sollen. Die Aufträge werden in Form eines Programmes an das Rechensystem gegeben, das üblicherweise nicht in einer maschinennahen Sprache, die direkt von einem Prozessor ausgeführt werden kann, formuliert ist. Es soll sichergestellt sein, daß nur solche Benutzer, die über einen berechtigten Zugang zum System verfügen, das Rechensystem benutzen können. Diese Anforderungen werden in Kapitel 7 im Rahmen der Prozeßverwaltung durch einen *Pförtner* gewährleistet.

Prozesse, durch die Aufträge ausgeführt werden, müssen erzeugt werden. Dies ist nur dann möglich, wenn genügend Speicherplatz zur Verfügung steht. Um die Berechnung ausführen zu können, muß ein Prozeß erzeugt und in das bestehende System so integriert werden, daß er den Prozessor und alle weiteren Betriebsmittel in Verbindung mit den Verwaltungskomponenten des Betriebssystems nutzen kann. Dies wird durch die Prozeßverwaltung in Verbindung mit der Speicherverwaltung gewährleistet. Ein Prozeß gehört zu einer Gruppe von Prozessen. Er wird entsprechend seiner Zugehörigkeit in die Menge der bereits existierenden Prozesse integriert.

Die hier für die Prozeßverwaltung beschriebenen Aufgaben werden in der Literatur mit den Beschreibungen des Prozeßkonzepts und deren Erzeugung bzw. Terminierung vorgestellt, siehe beispielsweise die Kapitel 3 in [Sta92], 4.3.1 und 4.3.2 in [GS94] oder 4.1.3 in [Spi95]. Der Zusammenschluß von Prozessen zu Gruppen ist eine Möglichkeit zur Beschreibung von Abhängigkeiten zwischen Prozessen, die insbesondere bei der Erzeugung von Kindprozessen entstehen; vergleiche auch hierzu die oben genannten Abschnitte. Die Hinzunahme eines Pförtners (*Login*) zum modellierten System sowie die Übersetzung eines Programms (*Compiler*) in maschinensprachlichen Code resultieren aus der Schnittstellenfunktion eines Betriebssystems; siehe Abschnitt 4.1.3 in [Spi95] sowie Kapitel 8.1 in [GS94]. Der Pförtner ist als stark vereinfachende Maßnahme zu den in Kapitel 14 in [GS94] oder Kapitel 10.2 in [Sta92] beschriebenen Verfahren zur Sicherheit in Rechensystemen zu sehen.

Kapitel 4

Prozessorverwaltung

Focus hat einen Reifegrad erreicht, in dem es zur prototypischen Entwicklung komplexer Systeme eingesetzt werden kann. Dabei ist zu zeigen, daß derartige Systeme mit den formalen Instrumentarien angemessen und mit vertretbarem Aufwand behandelbar sind. Wir haben uns für die Behandlung von Betriebssystemkonzepten als Anwendungsgebiet entschieden, vergleiche die Kapitel 1 und 3. Insgesamt verfolgen wir in der vorliegenden Arbeit vor allem zwei einander ergänzende Ziele:

1. Die Spezifikation der Betriebssystemkonzepte auf hohem Abstraktionsniveau und
2. den methodischen Einsatz der Focus-Instrumentarien.

Im vorliegenden Kapitel werden wir unter Verwendung der im Kapitel 2 definierten Schemata, der Beschreibungstechniken und formalen Konzepte von Focus und ausgehend von der in Kapitel 3 gegebenen Beschreibung der Betriebssystemkonzepte zunächst ein System zur Gewährleistung von Aufgaben der Prozessorverwaltung erarbeiten. Wir geben eine Einführung in die Aufgaben der Prozessorverwaltung mit Referenzen auf weiterführende Literatur und erläutern die methodische Vorgehensweise bei der Erstellung der formalen Modellierung. Anschließend wird schrittweise und systematisch die Spezifikation des Focus-Systems entwickelt, das das Kernsystem für alle weiteren in den Kapiteln 5 bis 7 entwickelten Modellierungen darstellt.

4.1 Einführung

Die Aufträge, die ein Benutzer an ein Rechensystem stellt, sind üblicherweise Berechnungen, die er unter Ausnutzung der Rechenleistung des Systems ausführen möchte. Das zentrale Konzept für den Umgang mit dieser Aufgabenstellung bilden die Betriebssystemprozesse, siehe beispielsweise [Tan92] oder [GS94]. *Prozesse* stehen für die Aufträge, die an das System gestellt werden, repräsentieren somit die Benutzer und bilden die Schnittstelle zu diesen. Prozesse sind die abstrakten, aktiven Komponenten innerhalb eines Betriebssystems, vergleiche Abschnitt 3.2.

- Ein Prozeß ist ein Strukturelement, das es erlaubt, die Beschreibung einer Berechnung (also das Programm) von deren Ausführung zu trennen. Er steht für ein Programm, das gestartet und dessen Ausführung noch nicht beendet ist.
- Die für die Durchführung von Berechnungen innerhalb eines Rechensystems und von einem Betriebssystem durchzusetzenden Verwaltungsaufgaben werden anhand des Prozeßkonzepts demonstriert.

Die Realisierung eines Prozesses erfolgt jedoch ausschließlich in Verbindung mit einem Prozessor. Ein- und Mehrprozessorsysteme bestimmen jeweils die Anzahl der Prozesse, die höchstens gleichzeitig aktiv sein dürfen. Die Ausführung der einem Prozeß zugeordneten Berechnung kann nur dann tatsächlich fortschreiten, wenn der Prozessor dem Prozeß zugeteilt ist. Da dies für alle Prozesse gilt, müssen sie sich das Betriebsmittel *Prozessor* teilen. Dabei muß dafür gesorgt werden, daß Prozesse den Prozessor überhaupt und dann nur für eine angemessene Zeitspanne zugeteilt bekommen.

Im Rahmen der Prozessorverwaltung besteht die Betriebssystemaufgabe darin, darüber zu entscheiden, welcher Prozeß als nächstes einen Prozessor zugeteilt bekommt, und dafür zu sorgen, daß der Prozessor wieder freigegeben wird. Die Entscheidung darüber, welcher Prozeß als nächstes ausgewählt wird, wird mittels eines *Scheduling*-Algorithmus getroffen. Für das Umschalten zwischen Prozessor und dem nächsten ausgewählten Prozeß sowie die Weitergabe des Prozessors ist der *Dispatcher* zuständig.

Scheduling

Der Teil eines Betriebssystems, der entscheidet, welchem ausführbereiten Prozeß der Prozessor als nächstes zugeteilt wird, heißt *Scheduler*. Im allgemeinen existieren im System mehrere Prozesse, deren Berechnungen ausgeführt werden sollen. Durch einen Scheduler wird ein Scheduling-Algorithmus realisiert und auf der Basis einer vorgegebenen Strategie werden ausführbereite Prozesse ausgewählt. In Abschnitt 3.3 wurden bereits einige Kriterien zur Charakterisierung von Scheduling-Algorithmen, wie Fairneß oder Effizienz, genannt. Scheduling-Verfahren, bei denen der Prozessor einem ausführbereiten Prozeß entzogen, der Prozeß *suspendiert* wird, werden als *Preemptives Scheduling* bezeichnet. Es gibt viele spezielle Scheduling-Algorithmen, deren Schwerpunkte entweder auf eines der möglichen Kriterien ausgerichtet sind oder einen Kompromiß zwischen mehreren unterschiedlichen Kriterien bilden. Die Vor- und Nachteile dieser Algorithmen werden ausführlich in der Betriebssystemliteratur behandelt, siehe die Abschnitte 2.4 in [Tan94], 4.3 in [PS85], 6.2 in [Sta92] oder auch 5.2 in [Spi95]. Einige Algorithmen seien hier genannt:

Prioritäts-Scheduling: Prozesse verfügen jeweils über eine zugeordnete Priorität, und der Prozeß mit der höchsten Priorität wird als nächstes ausgeführt; siehe die Abschnitte 2.4.2 in [Tan94] oder 4.3.4 in [PS85].

Shortest-Job-First: Unter der Voraussetzung, daß die Ausführzeiten der Prozesse bekannt sind, und mehrere ausführbereite Prozesse gleicher Priorität auf ihre Ausführung warten, wird der Prozeß mit der kürzesten Ausführzeit ausgewählt; siehe die Abschnitte 2.4.4 in [Tan94] und 4.3.3 in [PS85].

Ein weit verbreiteter Scheduling-Algorithmus heißt *Round-Robin*: Jedem Prozeß wird ein bestimmtes Zeitintervall (*Quantum*) zugeteilt, das für seine Ausführung, also die Kopplung an den Prozessor, gültig ist. Sobald das Quantum abgelaufen ist, wird der Prozeß suspendiert. Der Prozessor wird ihm entzogen und einem anderen Prozeß zugeteilt. Wird ein Prozeß blockiert, etwa weil er ein anderes, vom Prozessor verschiedenes Betriebsmittel für die Fortführung einer Berechnung benötigt, wird der Prozessor zu diesem Zeitpunkt freigegeben. Ebenso wird der Prozessor freigegeben, wenn die Ausführung seiner Berechnung terminiert, bevor das Quantum abgelaufen ist. Zur Durchsetzung dieses Algorithmus wird eine Liste ausführbereiter Prozesse verwaltet.

Zur Realisierung des Round-Robin-Verfahrens ist die Länge des Quantums geeignet zu bestimmen, da es weder zu klein noch zu groß gewählt werden darf. Ein Prozeß muß zum einen in jedem Fall mit der Ausführung seiner Berechnung vorankommen, und zum anderen muß gewährleistet sein, daß der Prozessor zwischen den Prozessen angemessen aufgeteilt wird. Auf derartige Details werden wir nicht eingehen, da sie in unserer Modellierung keine Rolle spielen. Zur Bestimmung der geeigneten Länge des Quantums und für weitere Erklärungen zum Round-Robin-Verfahren verweisen wir u.a. auf die Abschnitte 2.4.1 in [Tan94], 4.3.6 in [PS85] oder auch 5.2.2 in [Spi95].

Im vorliegenden Kapitel modellieren wir ein System, in dem die Verwaltung der ausführbereiten Prozesse, deren Suspendierung und die Weitergabe des Prozessors mit Round-Robin gewährleistet sind. Auf die Modellierung verschiedener Schedulingalgorithmen gehen wir nicht ein. Als Erweiterung behandeln wir ein System mit mehreren Prozessoren, wobei wir alternativ eine zentrale und dezentrale Verwaltung aller Prozessoren modellieren.

4.2 Methodische Vorgehensweise

In Abschnitt 4.1 wurden die grundlegenden Aufgaben der Prozessorverwaltung erklärt. Unsere Spezifikationsentwicklung startet mit Formalisierung eines verteilten Focus-Systems, mit dem charakteristische Konzepte der Prozessorverwaltung formal modelliert werden. Die dabei erstellten Formalisierungen sind die Basis für alle folgenden Spezifikationen. Die formalen Modellierungen werden in einer Art und Weise erarbeitet, in der sie für Leser verständlich sind, die mit Focus bisher nur wenig vertraut sind. Wir werden uns dem geforderten komplexen Verhalten schrittweise und systematisch ausgehend von einer auf den wesentlichen Kern reduzierten Aufgabenstellung nähern. Mit dieser Vorgehensweise können wir uns in den folgenden Kapiteln auf die Lösung der zunehmenden komplexer werdenden Aufgabenstellung konzentrieren, ohne auf die formalen Techniken von Focus detaillierter einzugehen.

Wir entwickeln eine abstrakte Beschreibung von *Prozessen* und modellieren die Prozesse und deren Kernfunktionalität bezogen auf ihre Verwaltung und die Zuteilung eines Prozessors. Alle formalen Modellierungen werden es zudem ermöglichen, aufgrund der gewählten Abstraktion, den Begriff eines Prozesses und die in Betriebssystemen anfallenden Verwaltungsaufgaben sowie die dort durchzusetzenden Strategien unabhängig von einer konkreten Anwendung zu verstehen. Die Aufgabe der Prozessorverwaltung wird unabhängig von einer Implementierung verständlich dargestellt. Ein weiteres Ziel besteht darin, die verwendeten Beschreibungstechniken sowie die für FOCUS charakteristische Sicht zur Beschreibung verteilter Systeme anhand der Modellierung dieses Kernsystems verständlich zu machen. Die Aufgabenstellung lautet:

> Für ein System mit endlich vielen Prozessen wird die Verwaltung des exklusiv nutzbaren Betriebsmittels „Prozessor" modelliert.

Im ersten Schritt zur Entwicklung eines verteilten FOCUS-Systems, mit dem zentrale Betriebssystemkonzepte formal modelliert werden, gehen wir davon aus, daß ein Prozeß zur Ausführung der Berechnung ausschließlich den Prozessor und keine weiteren Betriebsmittel benötigt. Wir betrachten von dem in Abschnitt 3.2 und dort mit Abbildung 3.2.1 gezeigten Zustandsdiagramm für Prozesse zunächst nur die beiden Zustände *ready* und *busy*. Das verteilte System umfaßt die ausführbaren Prozesse, den Teil, der die Aufgaben der Prozessorverwaltung übernimmt, und den Prozessor. Wir erhalten die in Abbildung 4.2.1 gezeigte erste grobe Vorstellung der Struktur des zu modellierenden Systems.

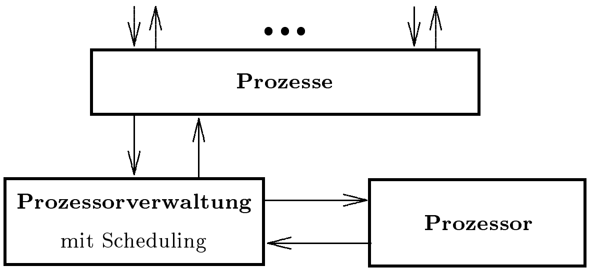


Abbildung 4.2.1: System mit Prozessorverwaltung

Als Einstiegspunkt zur Modellierung der Prozessorverwaltung gehen wir zunächst von der einfachst möglichen Konstellation aus. Im System gibt es einen Prozeß und einen Prozessor, den der Prozeß für die Durchführung seiner Berechnung benötigt. In Vorbereitung auf die später zu behandelnde Verwaltungsaufgabe ist eine weitere Komponente, der Dispatcher, für die Zuteilung des Prozessors und die Suspendierung des Prozesses zuständig. Diese erste Systemkonstellation wird in mehreren Schritten erweitert. Zunächst vervielfältigen wir die Anzahl der Prozesse. In einem weiteren Schritt wird die Anzahl der im System vorhandenen Prozessoren erhöht. Bei der Behandlung dieses Systems verfolgen wir zwei Alternativen: Die Verwaltung jedes Prozessors durch einen separaten Dispatcher und die Spezifikation eines zentralen Dispatchers, der alle Prozessoren verwaltet. Die erste Alternative entspricht einer abstrakten Beschreibung der Prozessorverwaltung in verteilten Betriebssystemen und die zweite einer Modellierung zentraler Mehrprozessorsysteme.

Zur Erstellung der Spezifikationen verwenden wir konsequent die in Kapitel 2 festgelegten Schemata. Wir geben eine informelle Beschreibung der Anforderungen an das betrachtete System. Aufbauend auf dieser Beschreibung erarbeiten wir die geforderte Funktionalität der einzelnen Komponenten. Das Verhalten jeder Komponente wird mittels einer strukturierten textuellen Darstellung festgelegt, die den Spezifikationsschemata aus Kapitel 2 zugrundeliegt. Die Erstellung der formalen Spezifikation in Form eines Gleichungssystems erfolgt abschließend durch die konsequente Umsetzung gemäß der Spezifikationsschemata.

4.3 Dispatching für einen Prozeß

Die hier zu behandelnde Aufgabenstellung stellt eine stark vereinfachte Konstellation bzgl. der für die formale Modellierung der Prozessorverwaltung geforderten Funktionalität dar und bildet die Basis für alle weiteren Modellierungen. Wir fordern:

> Für ein Einprozessorsystem werden die Zuteilung und die Freigabe des exklusiv nutzbaren Betriebsmittels „Prozessor" in einem System mit einem Prozeß modelliert.

Prozesse repräsentieren die Programme, die sich in Ausführung befinden. Prozesse sind die abstrakten, aktiven Komponenten, an der alle durch ein Betriebssystem durchzuführenden Verwaltungsaufgaben erklärt und demonstriert werden. Wir gehen von folgenden vereinfachenden Annahmen aus: Außer dem Prozessor benötigt der Prozeß zunächst keine weiteren Betriebsmittel. Für den Prozeß ist eine Folge von atomaren Berechnungsschritten (Instruktionen) definiert, die eine abstrakte Darstellung des auszuführenden Programms liefert. Die Rechenschritte seien in Maschinensprache formuliert. Jeder Schritt muß in Verbindung mit dem Prozessor ausgeführt werden, der die Instruktionen als einzige Komponente des Systems tatsächlich ausführen kann. Sobald das Programm abgearbeitet ist, terminiert der Prozeß. Wir setzen voraus, daß der Prozeß nur zu Beginn mit der Eingabe seines Programms durch die Umgebung gestartet wird und nicht mehr mit der Umgebung interagiert.

Im Hinblick darauf, daß die hier gezeigte Spezifikation den ersten Schritt zur später gezeigten Modellierung der Prozessorverwaltung für n Prozesse darstellt, gehört zu dem System eine Komponente, die für das Umschalten des Prozessors zwischen den Prozessen zuständig ist. Der Dispatcher übernimmt die Zuteilung des Prozessors an einen anfordernden Prozeß sowie die Suspendierung des Prozesses, sobald der Prozessor dem Prozeß entzogen werden muß. Prozesse verfügen nicht über unterschiedliche Prioritäten, sondern werden bei der Zuteilung des Prozessors gleich behandelt. Wir verzichten auf die Modellierung verschiedener Schedulingstrategien und modellieren ein System, in dem eine Round-Robin-Strategie realisiert wird. Der Prozessor wird einem Prozeß für eine festgelegte Zeitspanne zugeteilt. Dieser *Dispatcher* sorgt dafür, daß der Prozessor jedem Prozeß, der sich um ihn beworben hat, schließlich zugeteilt wird. Zusätzlich hat der Dispatcher die Aufgabe, die Einhaltung der zugeteilten Zeitspanne zu gewährleisten. Dies erfolgt dadurch, daß er bei Zuteilung des Prozessors einen Zeitgeber mit einem vorgegebenen Zeitlimit startet und die Suspendierung des Prozesses veranlaßt, sobald die Zeitspanne abgelaufen ist. Ein Prozeß, der den Prozessor benötigt, wird in einen Warteraum, eine *Queue*, eingetragen, aus dem der Dispatcher den nächsten Prozeß für die Zuteilung des Prozessors entnimmt.

Die zentrale Idee besteht darin, das System als mobiles, dynamisches FOCUS-Netz zu modellieren, in dem die Kopplung von Prozeß und Prozessor durch den Aufbau direkter Kanalverbindungen zwischen beiden Komponenten hergestellt wird. Der Dispatcher initiiert den Aufbau dieser Verbindung. In den nächsten Abschnitten zeigen wir zunächst das verteilte System und die Verknüpfung der genannten Komponenten, siehe Abschnitt 4.3.1, und spezifizieren die Komponenten in den Abschnitte 4.3.3 bis 4.3.7 jeweils separat.

4.3.1 Ein erstes verteiltes System zur Prozessorverwaltung

Mit den oben gegebenen Erläuterungen ergibt sich ein verteiltes System, das aus den Komponenten *Prozeß*, *Prozessor*, *Dispatcher*, *Queue* und *Timer* besteht. Das SSD zu diesem System `OneP` zeigt Abbildung 4.3.1. Die Verbindung von `OneP` zur Umgebung wird durch den Prozeß und dessen Lese- bzw. Schreibverbindung über die Kanäle In_1 bzw. Out_1 definiert. Die Verbindungen der Komponenten untereinander werden in den später gezeigten Spezifikationen erklärt. Das SSD zeigt zwei Systemphasen: In *Phase 0* ist $P1$ *nicht* an PZ gebunden. In *Phase 1* ist $P1$ durch die Kanäle *PtoPZ* und *PZtoP* mit PZ verbunden. *Phase 0* zeigt repräsentativ die Systemstruktur dafür, daß der Prozeß im Zustand *ready* ist: er ist bereit, seine Berechnung auszuführen, aber das Betriebsmittel *Prozessor* steht für ihn aktuell nicht zur Verfügung. *Phase 1* zeigt die Systemstruktur, falls der Prozeß im Zustand *busy* ist, und die Ausführung der Berechnung aktuell voranschreitet.

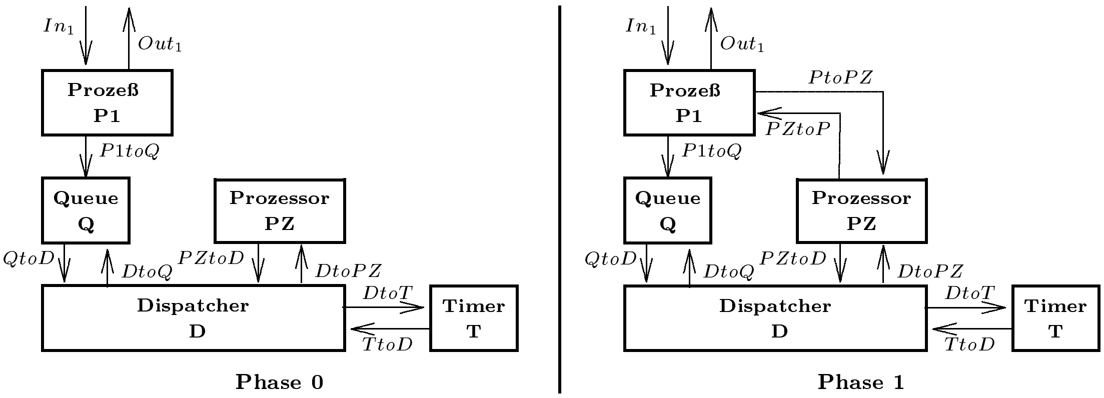


Abbildung 4.3.1: Das Einprozessorsystem `OneP`

Die Umsetzung des in Abbildung 4.3.1 gezeigten SSD in eine ANDL-Spezifikation gemäß Abschnitt 2.8 liefert die in Abbildung 4.3.2 gezeigte Spezifikation. Die zugeordneten Nachrichtentypen werden in den folgenden Abschnitten festgelegt. Mit einer ANDL-Spezifikation wird die initiale Vernetzung eines mobilen, dynamischen Netzes angegeben.

agent `OneP`
  input channels $In_1 : S_{In_1}$
  output channels $Out_1 : S_{Out_1}$
is network

$$
\begin{array}{lcl}
\ll Out_1, P1toQ \gg & = & \mathtt{P1} \ll In_1 \gg \;; \\
\ll QtoD \gg & = & \mathtt{Q} \ll P1toQ, DtoQ \gg \;; \\
\ll DtoQ, DtoT, DtoPZ \gg & = & \mathtt{D} \ll QtoD, TtoD, PZtoD \gg \;; \\
\ll PZtoD \gg & = & \mathtt{PZ} \ll DtoPZ \gg \;; \\
\ll TtoD \gg & = & \mathtt{T} \ll DtoT \gg \;;
\end{array}
$$

end `OneP`

Abbildung 4.3.2: ANDL-Spezifikation für das System `OneP`

N_{OneP} sei die Menge der im System `OneP` definierten Kanalbezeichner. Die Bezeichner können in den gezeigten ANDL-Spezifikationen abgelesen werden.

4.3.2 Zusammenspiel der Komponenten

Zum Verständnis des geforderten Verhaltens ist eine graphische Darstellung des Zusammenspiels der Komponenten oftmals hilfreich. In Abschnitt 2.7 wurden die EETs zur graphischen Beschreibung derartiger Systemsichten erklärt. Wir zeigen die EETs für einen Prozeß, dessen Berechnung durch $\langle Step_1, Step_2, Step_3 \rangle$ repräsentiert ist.

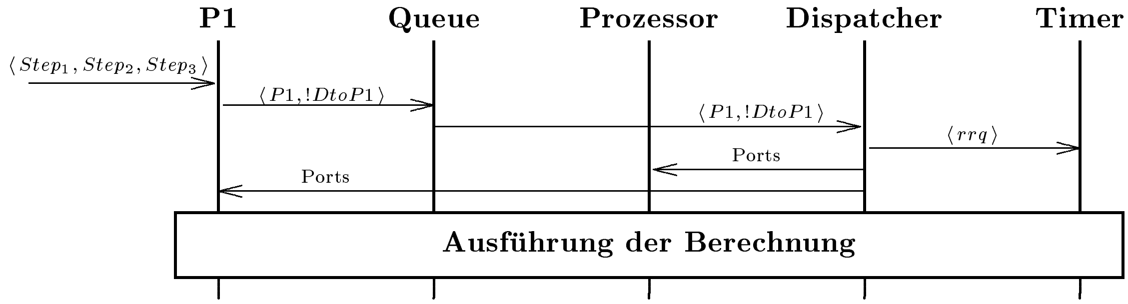
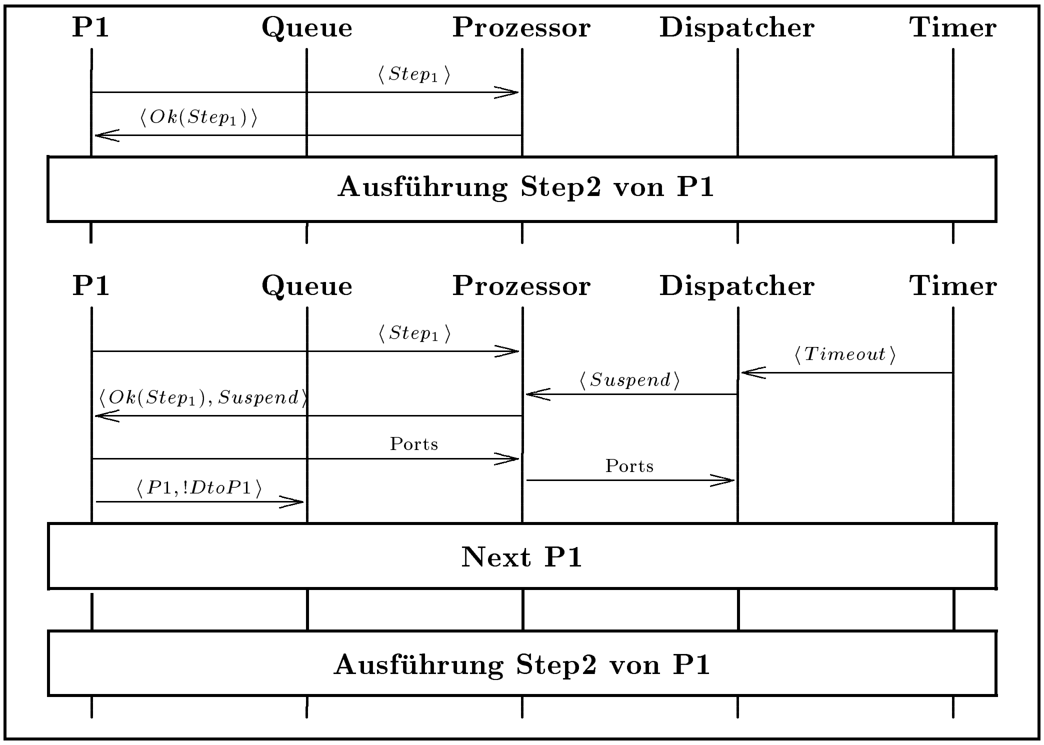


Abbildung 4.3.3: EET zur Prozessorverwaltung mit einem Prozeß

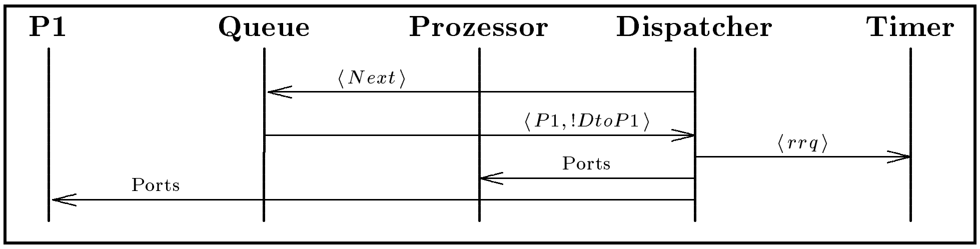
Das in Abbildung 4.3.3 gezeigte hierarchische EET beschreibt das folgende Verhalten: $P1$ wird mit $\langle Step_1, Step_2, Step_3 \rangle$ gestartet. Da $P1$ den Prozessor benötigt, fordert er diesen an, indem er sich in die Queue Q einträgt. Q ist zu Beginn leer, die Anforderung des Prozessors wird direkt an den Dispatcher D weitergegeben. Die Zuteilung des Prozessors erfolgt durch D, der zudem den Timer mit dem Quantum rrq startet. $P1$ und der Prozessor werden durch die Erzeugung von Kanalverbindungen und das Versenden entsprechender Ports gekoppelt. In den hier gezeigten EETs wird eine Sequenz von Ports durch *Ports* abgekürzt. Die jeweils gültige Sequenz kann in den später gezeigten Spezifikationen nachgelesen werden. Sind Prozessor und Prozeß verbunden, wird die Berechnung ausgeführt.

Ausführung der Berechnung



Die Ausführung der Berechnung startet damit, daß der Prozeß $Step_1$ an den Prozessor sendet. Zur Fortsetzung der Berechnung unterscheiden wir folgende Fälle: Entweder die Zeitscheibe rrq ist noch nicht abgelaufen, oder rrq ist abgelaufen, und die Kopplung zwischen Prozessor und Prozeß wird gelöst. $P1$ bewirbt sich erneut um den Prozessor. Somit besteht die Box *Ausführung der Berechnung* aus zwei alternativen EETs.

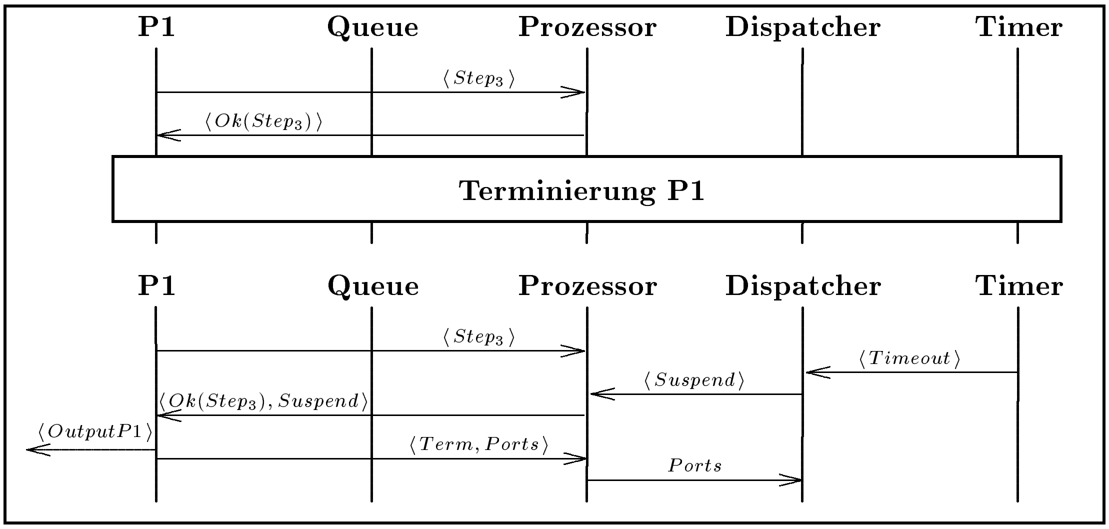
Next P1



Die Box *Next P1* enthält ein EET und dient als Abkürzung. Der Dispatcher fordert bei Q den nächsten ausführbereiten Prozeß an, und $P1$ ist der ausgewählte Prozeß. Der Timer wird mit rrq gestartet und die Kanalverbindung zwischen Prozeß und Prozessor hergestellt.

Der nächste Schritt besteht in der Ausführung von $Step_2$ und wird mit der Box *Ausführung Step2 von P1* dargestellt. Sie enthält zwei alternative EETs und entspricht dem EET zur Ausführung von $Step_1$, wobei $Step_1$ durch $Step_2$ ersetzt und statt der Box *Ausführung von Step1* eine Box *Ausführung von Step3* verwendet wird; diese wird im folgenden erklärt. Wir verzichten auf die graphische Darstellung der Box zur Ausführung von $Step_2$.

Ausführung Step3 von P1



Als nächstes ist $Step_3$ auszuführen. Dieser Ablauf wird durch die Box *Ausführung Step3 von P1* dargestellt, die zwei alternative EETs enthält. Zunächst erfolgt die Ausführung von $Step_3$ entsprechend zur Ausführung von $Step_1$ und $Step_2$. Da $Step_3$ jedoch der letzte auszuführende Berechnungsschritt ist, terminiert die Berechnung abschließend. Bei der Terminierung von $P1$ werden folgende Fälle unterschieden:

1. Die Zeitscheibe ist abgelaufen, während $Step_3$ ausgeführt wird. $P1$ wird suspendiert, die Bindungen zwischen Prozeß und Prozessor werden gelöst und *OutputP1* an die Umgebung gesendet.

2. Die Berechnung wurde beendet, bevor rrq abgelaufen ist. $P1$ löst die Bindung zum Prozessor und sendet *OutputP1* an die Umgebung. Der Timer wird gestoppt.

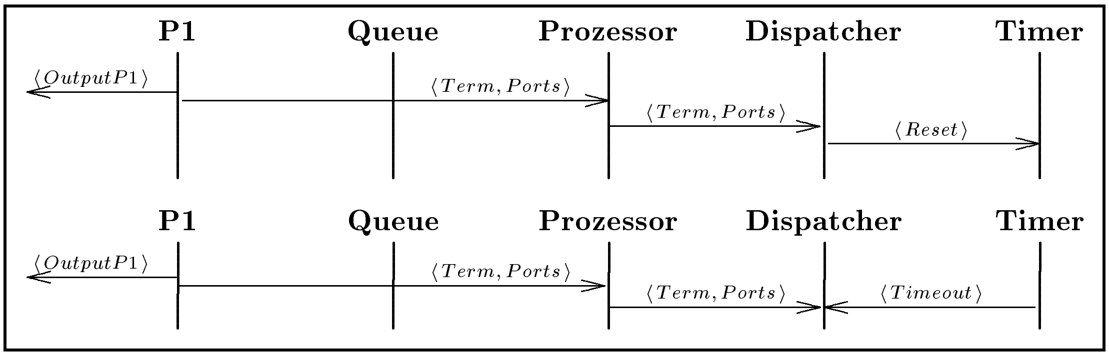
3. Die Berechnung wurde beendet, bevor das Quantum abgelaufen ist. $P1$ sendet *OutputP1* an die Umgebung. Während die Bindung zum Prozessor gelöst wurde, ist rrq abgelaufen. Der Timer wird nicht gestoppt.

Fall 1. ist in der Box *Ausführung Step3 von P1* dargestellt. Die Fälle 2. und 3. werden durch die Box *Terminierung von P1* veranschaulicht, die entsprechend zwei alternative EETs enthält.

Insgesamt zeigen diese graphischen Darstellungen den Ausschnitt aus dem in der Literatur gezeigten Prozeßzustandsdiagramm, siehe auch Abbildung 3.2.1, der den Wechsel zwischen den Zuständen *ready* und *busy* charakterisiert. Die entsprechenden Phasen treten in Form alternativer EETs bei der Ausführung der Berechnungsschritte auf. Das Verhalten einer

Komponente wird durch ein EET jedoch noch nicht eindeutig bestimmt. So ist das hier dargestellte Verhalten einer Verwaltungskomponente beispielsweise die des Dispatchers zu verallgemeinern. Der Dispatcher ist dafür zuständig, die korrekte Kopplung zwischen anforderndem Prozeß und Prozessor bei jeder Anforderung zu gewährleisten.

Terminierung P1



4.3.3 Der Prozeß

Bei der Spezifikation eines Prozesses muß berücksichtigt werden, daß die Aufträge der Umgebung, die letztlich durch die Benutzer des Rechensystems repräsentiert ist, durch Prozesse ausgeführt werden. Damit bilden die Prozesse die Verbindung zwischen dem Betriebssystem und den Benutzern. Wie die Ausführung der Aufträge organisiert wird, welche Betriebsmittel für die Ausführung benötigt werden, und wie die um diese Betriebsmittel konkurrierenden Prozesse zur Ausführung gebracht werden, ist für den Benutzer uninteressant. Der Benutzer liefert seinen Auftrag an das System und erwartet das Ergebnis.

Ein *Prozeß* $P1$ ist mit der Umgebung durch den Eingabekanal In_1 und den Ausgabekanal Out_1 verbunden. Zur Ausführung des Auftrags, für den der Prozeß zuständig ist, benötigt er in unserem ersten einfachen System das Betriebsmittel *Prozessor*. Der Dispatcher ist die Komponente, die den Prozessors verwaltet. $P1$ muß in der Lage sein, seine Anforderung nach Zuteilung des Prozessors in den Warteraum des Dispatchers einzufügen. Hierfür definieren wir den Kanal $P1toQ$, den $P1$ schreibend nutzen kann.

Wir haben bereits erklärt, daß die betrachtete Aufgabenstellung als Netz modelliert wird, in dem die Zuteilung des Betriebsmittels durch die zeitweise Kopplung der Komponenten erfolgt. Ausgehend von dieser Vorstellung ist es notwendig, daß $P1$ vom Dispatcher die Ports für den Verbindungsaufbau zum Prozessor empfangen kann. Hierfür verfügt $P1$ über einen privaten Kanal $DtoP1$. Mit der Anmeldung beim Dispatcher gibt $P1$ auch das Schreibrecht $!DtoP1$ weiter. Wird $P1$ als nächster Prozeß ausgewählt, erhält der Dispatcher auch das Schreibrecht $!DtoP1$ und kann auf diesem Weg die Kanalverbindungen zwischen Prozeß und dem Prozessor erzeugen. Die in Abbildung 4.3.4 gezeigten SSDs veranschaulichen die Schnittstellen von $P1$ in den beiden Phasen, die dem Wechsel zwischen den Zuständen *ready* und *busy* entsprechen. Auf die Beschreibung der Nachrichtentypen, die für die Kanäle gültig sind, verzichten wir zunächst.

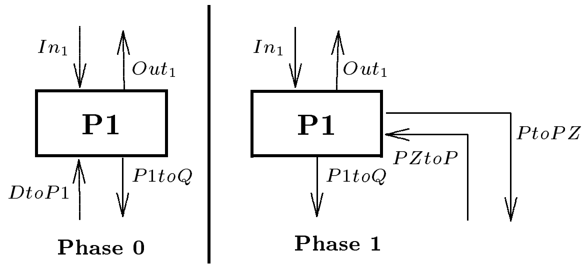


Abbildung 4.3.4: SSDs zu den Phasen *ready* und *busy*

Aus diesen Erläuterungen ergibt sich für $P1$ die Schnittstelle bestehend aus der Menge der *aktiven* Ports ap_{P1} und der Menge der *privaten* Ports pp_{P1}. Die in Abbildung 4.3.4 gezeigte *Phase 0* charakterisiert alle Phasen, in denen $P1$ im Zustand *ready* ist.

$$ap_{P1} = \{!Out_1, !P1toQ\} \ \cup \ \{?In_1, ?DtoP1\} \quad \wedge \quad pp_{P1} = \emptyset$$

Diese ändert sich in *Phase 1* zu

$$ap_{P1} = \{!Out_1, !P1toQ, !PtoPZ\} \ \cup \ \{?In_1, ?PZtoP\} \quad \wedge \quad pp_{P1} = \{?!DtoP1\}$$

Für die Zeitspanne, in denen sich der Prozeß beim Dispatcher angemeldet hat, ihm der Prozessor jedoch noch nicht zugeteilt wurde, ist die Menge pp_{P1} leer, und der Port $?DtoP1$ gehört zur aktuellen Schnittstelle ap_{P1}.

Kanal n	Nachrichtenmengen S_n	
In_1	$STEP$	$\cup\ ?!N_{OneP}$
Out_1	$\{OutputP1\}$	
$P1toQ$	$\{P1\}$	
$DtoP1$	$\emptyset$	

Tabelle 4.3.1: Nachrichtentypen für P1

Bevor wir die Spezifikation von $P1$ mittels einer ANDL-Spezifikation, siehe Abschnitt 2.8, erstellen, legen wir die Nachrichtentypen der genannten Kanälen fest. $P1$ wird mit dem *Programm* gestartet, das die Berechnung repräsentiert. $P1$ fordert den Prozessor an, indem er seinen Identifikator in die Warteschlange einträgt. Bei der Ausführung des Programms werden die einzelnen Berechnungsschritte sukzessive an den Prozessor gesendet. Wird $P1$ suspendiert solange die Berechnung noch nicht beendet ist, bewirbt er sich erneut beim Dispatcher um den Prozessor. Sobald die Berechnung ausgeführt ist, wird die Ausgabe *OutputP1* an die Umgebung gesendet, und $P1$ terminiert. Die abstrakten Nachrichten, die zur Beschreibung dieses Verhaltens verwendet werden, sind in Tabelle 4.3.1 aufgelistet. N_{OneP} steht für die Menge der Kanalbezeichner im System `OneP`, siehe Abschnitt 4.3.1.

Mit $STEP = \{Step_i \mid i \in I\!N\}$ werden die vom Prozessor direkt ausführbaren Berechnungsschritte abgekürzt. Basierend auf Abbildung 4.3.4 und Tabelle 4.3.1 geben wir die ANDL-Spezifikation der Basiskomponente $P1$ an.

agent P1
input channels $In_1 \ : \ S_{In_1}$
output channels $Out_1 \ : \ S_{Out_1}, \ P1toQ \ : \ S_{P1toQ}$
private channels $DtoP1 \ : \ S_{DtoP1}$
is basic
$\boldsymbol{f_{P1}}$ $\boxed{\text{mit der Spezifikation von Seite 57}}$
end P1

Abbildung 4.3.5: ANDL-Spezifikation von $P1$

Entsprechend zur Beschreibung von Prozessen in der Literatur, siehe beispielsweise [PS85] oder [Tan94], werden wir $P1$ zustandsorientiert spezifizieren. Das Zustandsdiagramm für Prozesse wurde bereits in Abschnitt 3.2 mit Abbildung 3.2.1 gezeigt. Den Ausschnitt, der für das hier geforderte Verhalten wesentlich ist, zeigt Abbildung 4.3.6. Der Zustand *waiting* ist in diesem Diagramm nicht aufgeführt, da $P1$ zunächst außer dem Prozessor kein weiteres Betriebsmittel benötigt, um seine Berechnung durchzuführen. Ebenso verzichten wir auf den Zustand *terminated*, auf den wir im Kapitel 7 eingehen werden.

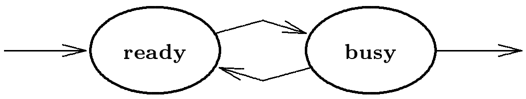


Abbildung 4.3.6: Vereinfachtes Zustandsdiagramm für Prozesse

Zur Spezifikation von $P1$ verwenden wir die Zustandsmenge $State_{P1} = \{ready, busy, term\}$

Zur Festlegung des Verhaltens von $P1$ und somit zur Spezifikation der Funktion f_{P1} müssen die Anforderungen an den Prozeß präzisiert werden. Diese werden in einer strukturierten textuellen Form so angegeben, daß sie mit den in Kapitel 2.1 vorgegebenen Spezifikationsschemata umgesetzt werden können. Mit p bezeichnen wir die aktuell gültige Sequenz der Berechnungsschritte.

(1) Erhält $P1$ über Kanal In_1 die Eingabe $\langle Step_1, \ldots, Step_n \rangle$, sendet er die Nachricht $P1$ und den Port $!DtoP1$ über Kanal $P1toQ$ und geht in den Zustand *ready* über.

(2) Erhält $P1$ im Zustand *ready* über Kanal $DtoP1$ die Ports $?PZtoP$, $!PtoPZ$ und $!DtoP1$, sendet er über Kanal $PtoPZ$ den als nächstes auszuführenden Berechnungsschritt $ft.p$ und geht in den Zustand *busy* über .

(3) Erhält $P1$ im Zustand *busy* die Nachricht $Ok(ft.p)$ über Kanal $PZtoP$, so sendet er den nächsten Berechnungsschritt $ft.rt.p$ über Kanal $PtoPZ$ und verbleibt im Zustand *busy*.

(4) Erhält $P1$ im Zustand *busy* die Nachrichten $Ok(ft.p)$ und $Suspend$, werden die Verbindungen zum Prozessor gelöscht. $P1$ sendet $\langle P1, !DtoP1 \rangle$ über Kanal $P1toQ$ und geht in den Zustand *ready* über.

(5) Die Schritte (2), (3) und (4) werden wiederholt, bis die Berechnung beendet ist. Erhält $P1$ das „Ok" zum letzten Berechnungsschritt, sendet er die Nachricht *Term* über Kanal $PtoPZ$ und löscht die Verbindungen zu PZ. Zusätzlich sendet $P1$ ein $\langle OutputP1 \rangle$ über Kanal Out_1 und geht in den Zustand *term* über.

Die Umsetzung der textuellen Beschreibung liefert die folgende Spezifikation.

Funktionsgleichungen für f_{P1}

$\forall s \in \prod_{n \in N_{OneP}} [S_n^*],\ Step_1, \ldots, Step_n \in STEP,\ p \in STEP^*$:
$\exists h \in (State_{P1} \times STEP^*) \rightarrow Type_{P1}$:

(1) $f_{P1}(\{In_1 \mapsto \langle Step_1, \ldots, Step_n \rangle\} \circ s)$
$= \{P1toQ \mapsto \langle P1, !DtoP1 \rangle\} \circ h(ready, Step_1, \ldots, Step_n)(s)$

(2) $h(ready, p)(\{DtoP1 \mapsto \langle ?PZtoP, !PtoPZ, !DtoP1 \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle ft.p \rangle\} \circ h(busy, p)(s)$

Für $\#p > 1$:

(3) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p) \rangle\} \circ s) = \{PtoPZ \mapsto \langle ft.rt.p \rangle\} \circ h(busy, rt.p)(s)$

(4) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p), Suspend \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle ?PZtoP, !PtoPZ \rangle, P1toQ \mapsto \langle P1, !DtoP1 \rangle\} \circ h(ready, rt.p)(s)$

Für $\#p = 1$:

(5a) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p) \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ \rangle, Out_1 \mapsto \langle OutputP1 \rangle\} \circ h(term, \langle\rangle)(s)$

(5b) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p), Suspend \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ \rangle, Out_1 \mapsto \langle OutputP1 \rangle\} \circ h(term, \langle\rangle)(s)$

Die Gleichungen (1) und (3) ergeben sich direkt gemäß dem allgemeinen Schema (2.7) von Seite 20 für pulsgetriebene stromverarbeitende Funktionen. Zusätzlich gilt allgemein die Konvention (2.8), so daß wir beispielsweise darauf verzichten können, in (2) bis (5b) den Eingabekanal In_1 explizit zu berücksichtigen. Die Forderung (5) wird durch zwei Gleichungen spezifiziert, die die Fälle repräsentieren, die hier auftreten können: Zum einen wird ausschließlich das Berechnungsergebnis und zum anderen das Ergebnis gemeinsam mit der

Aufforderung zur Suspendierung empfangen. Mit der Gleichung (2) erhält $P1$ Zugriffsrechte zu neuen Kanalverbindungen, dieser Verbindungsaufbau wurde durch eine andere Komponente initiiert. Mit den Gleichungen (4), (5a) und (5b) initiiert $P1$ das Löschen von Kanälen, die zur aktuellen Schnittstelle ap_{P1} gehören. Die Umsetzung dieser Forderung erfolgt systematisch gemäß Schema (2.15) von Seite 27. Die Sequenzen $\langle Ok(ft.p)\rangle$ bzw. $\langle Ok(ft.p), Suspend\rangle$ werden für den im Schema verwendeten Platzhalter *delete* eingesetzt. Auf die Modellierung von *term* werden wir in Kapitel 7 eingehen.

4.3.4 Der Prozessor

Der Prozessor PZ repräsentiert im hier betrachteten System das einzige zu verwaltende Betriebsmittel. Nur in Verbindung mit dem Prozessor kann die Berechnung, die einen Prozeß charakterisiert, aus- bzw. weitergeführt werden. Der Dispatcher ist die Komponente, die den Prozessor verwaltet und für die Zuteilung des Prozessors sowie die Suspendierung des Prozesses zuständig ist. Der Prozessor ist permanent mit dem Dispatcher sowohl durch eine Verbindung mit Lesezugriff, hier $DtoPZ$, als auch durch eine Verbindung mit Schreibzugriff, hier $PZtoD$ verbunden. Über Kanal $PZtoD$ meldet der Prozessor, wenn ein Prozeß den Prozessor freigibt, bevor die Zeitscheibe abgelaufen ist. Die Zuteilung des Prozessors an einen Prozeß wird mittels der Kanalverbindungen $PZtoP$ mit Schreib- und $PtoPZ$ mit Lesezugriff für den Prozessor modelliert. Der Prozessor durchläuft somit zwei Phasen: In *Phase 0* ist er frei, also an keinen Prozeß gebunden. In *Phase 1* ist er an einen Prozeß gebunden. Abbildung 4.3.7 zeigt die Schnittstelle von PZ für beide Phasen.

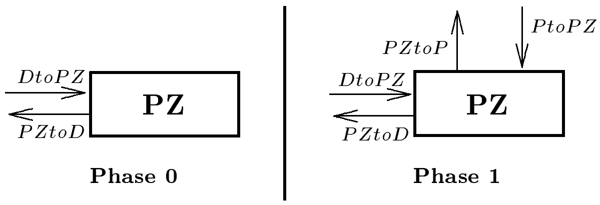


Abbildung 4.3.7: SSDs für die Phasen von PZ

Wir legen fest, daß die Menge pp_{PZ} leer ist und PZ eigenständig keine neuen Kanäle erzeugen kann. Die initiale Vernetzung des Prozessors in *Phase 0* wird durch

$$ap_{PZ} \;=\; \{!PZtoD\} \;\cup\; \{?DtoPZ\} \;\wedge\; pp_{PZ} = \emptyset$$

definiert. Diese Schnittstelle gilt für alle Phasen, in denen der Prozessor an keinen Prozeß gebunden ist. Für die aktuelle Schnittstelle in *Phase 1* gilt

$$ap_{PZ} \;=\; \{!PZtoD, !PZtoP\} \;\cup\; \{?DtoPZ, ?PtoPZ\}$$

Zur Festlegung der Nachrichtenmengen für die genannten Kanäle betrachten wir das Verhalten detaillierter: Empfängt PZ über Kanal $DtoPZ$ die Ports $!PZtoP$ und $?PtoPZ$, so ist ihm ein Prozeß zugeteilt. PZ ist bereit, Instruktionen aus der Menge $STEP$, siehe Abschnitt 4.3.3, durchzuführen. Erhält PZ über Kanal $DtoPZ$ die Nachricht *Suspend*, sind die Verbindungen zum Prozeß zu lösen. Erhält PZ über Kanal $PtoPZ$ die Nachricht *Term*, so teilt PZ dem Dispatcher über Kanal $PZtoD$ mit, daß der Prozessor freigegeben wurde. Die Nachrichtentypen, für die PZ zugeordneten Kanäle sind in Tabelle 4.3.2 aufgelistet. Die Mengen S_{PZtoP} und S_{PtoPZ} werden in Abschnitt 4.3.5 in Tabelle 4.3.3 festgelegt.

Kanal n	Nachrichtenmengen S_n
$DtoPZ$	$\{Suspend\}$ $\cup$ $?!N_{OneP}$
$PZtoD$	$\{Term\}$ $\cup$ $?!N_{OneP}$

Tabelle 4.3.2: Nachrichtentypen für PZ

Mit diesen Informationen kann das SSD zu PZ direkt in die in Abbildung 4.3.8 gezeigte ANDL-Spezifikation umgesetzt werden.

```
agent PZ
  input channels    DtoPZ : S_DtoPZ
  output channels   PZtoD : S_PZtoD
  private channels  ∅
is basic
     f_PZ   [mit der Spezifikation von Seite 60]
end PZ
```

Abbildung 4.3.8: ANDL-Spezifikation von PZ

Wir beschreiben nun das für den Prozessor geforderte Verhalten systematisch in textueller Form, die gemäß der Schemata in Funktionsgleichungen umgesetzt werden kann.

(1) Erhält PZ über Kanal $DtoPZ$ die Ports $?PtoPZ$ und $!PZtoP$, ist der Prozessor an einen Prozeß gebunden und bereit, dessen Berechnung auszuführen.

(2) Erhält PZ über Kanal $PtoPZ$ eine Nachricht $Step \in STEP$ und kein *Suspend* über Kanal $DtoPZ$, sendet er die Nachricht $Ok(Step)$ über Kanal $PZtoP$.

(3) Erhält PZ über Kanal $DtoPZ$ die Nachricht *Suspend* und über Kanal $PtoPZ$ ein *Step*, wird $\langle Ok(Step_i), Suspend\rangle$ über Kanal $PZtoP$ gesendet.

(4) Erhält PZ über Kanal $PtoPZ$ die Ports $?PZtoP$ und $!PtoPZ$, so werden diese Ports zusammen mit den Ports $!PZtoP$ und $?PtoPZ$ über Kanal $PZtoD$ gesendet.

(5) Empfängt PZ die Nachricht $Term$ und die Ports $?PZtoP$ sowie $!PtoPZ$ über Kanal $PtoPZ$ und kein *Suspend* über Kanal $DtoPZ$, sendet er die Nachricht $Term$ und die Ports $?!PZtoP$ sowie $?!PtoPZ$ über Kanal $PZtoD$.

(6) Wenn PZ die Nachricht $Term$ und die Ports $?PZtoP$ sowie $!PtoPZ$ über Kanal $PtoPZ$ und über Kanal $DtoPZ$ die Nachricht *Suspend* empfängt, werden die Ports $?!PtoPZ$ und $?!PZtoP$ über Kanal $PZtoD$ gesendet.

Die Umsetzung liefert die folgende FOCUS-Spezifikation:

Funktionsgleichungen für f_{PZ}

$\forall s \in \prod_{n \in N_{OneP}} [S_n^*],\ Step \in STEP\ :$

(1) $f_{PZ}(\{DtoPZ \mapsto \langle !PZtoP, ?PtoPZ \rangle\} \circ s) \ = \ f_{PZ}(s)$

(2) $f_{PZ}(\{PtoPZ \mapsto \langle Step \rangle, DtoPZ \mapsto \langle \rangle\} \circ s) \ = \ \{PZtoP \mapsto \langle Ok(Step) \rangle\} \circ f_{PZ}(s)$

(3) $f_{PZ}(\{PtoPZ \mapsto \langle Step \rangle, DtoPZ \mapsto \langle Suspend \rangle\} \circ s)$
$\quad = \ \{PZtoP \mapsto \langle Ok(Step), Suspend \rangle\} \circ f_{PZ}(s)$

(4) $f_{PZ}(\{PtoPZ \mapsto \langle ?PZtoP, !PtoPZ \rangle\} \circ s) \ = \ \{PZtoD \mapsto \langle ?!PtoPZ, ?!PZtoP \rangle\} \circ f_{PZ}(s)$

(5) $f_{PZ}(\{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ \rangle, DtoPZ \mapsto \langle \rangle\} \circ s)$
$\quad = \ \{PZtoD \mapsto \langle Term, ?!PtoPZ, ?!PZtoP \rangle\} \circ f_{PZ}(s)$

(6) $f_{PZ}(\{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ \rangle, DtoPZ \mapsto \langle Suspend \rangle\} \circ s)$
$\quad = \ \{PZtoD \mapsto \langle ?!PtoPZ, ?!PZtoP \rangle\} \circ f_{PZ}(s)$

Zur Durchführung der abstrakten Berechnungsschritte $Step \in STEP$ gehen wir davon aus, daß sie immer vom Prozessor ausgeführt werden. Dies bedeutet insbesondere, daß die Erfolgsmeldung $Ok(Step)$ auch dann gemeldet wird, wenn PZ über den Kanal $DtoPZ$ eine Suspendierungsnachricht empfangen hat. Auf diese Weise sorgen wir dafür, daß die durch den Prozeß ausgeführte Berechnung immer um mindestens einen Schritt voranschreitet, wenn der Prozeß über den Prozessor verfügt. Mit (5) erfassen wir den Fall, daß der Prozeß terminiert, bevor die zugeordnete Zeitscheibe abgelaufen ist, und mit (6) die Möglichkeit, daß das Terminieren des Prozesses mit dem Ende der Zeitscheibe zusammenfällt.

Die Gleichungen (1), (2) und (3) sind eine direkte Umsetzung des allgemeinen Schemas (2.7) von Seite 20. Für die Gleichungen (2) und (5) ist die Konvention (2.8) von Seite 20 nicht passend, da hier explizit unterschieden werden muß, ob auf Kanal $DtoPZ$ die Nachricht *Suspend* empfangen wird, oder nicht. Mit (4), (5) sowie (6) werden sowohl ein Weitergeben von empfangenen Ports als auch das Löschen von Kanalverbindungen der eigenen aktuellen Schnittstelle spezifiziert. Diese Gleichungen ergeben sich durch Kombination der Schemata (2.14) von Seite 27 und (2.15) von Seite 27.

4.3.5 Der Dispatcher

Der Dispatcher D ist die zentrale Komponente für Verwaltungsaufgaben im System `OneP`. D veranlaßt die Zuteilung des Prozessors und die Suspendierung eines Prozesses und registriert die erfolgreiche Suspendierung oder die Terminierung des Prozesses. D ist permanent mit der Queue, dem Timer und dem Prozessor verbunden. Zur Queue Q verfügt D über die Kanäle $DtoQ$, den er lesend, und $QtoD$, den er schreibend nutzen kann. Diese Verbindung wird benötigt, da D den Identifikator des nächsten auf den Prozessor wartenden Prozesses aus der Queue entnimmt. Der Dispatcher muß in der Lage sein, den Zeitgeber zu starten und die Mitteilung entgegenzunehmen, daß die Zeitscheibe abgelaufen ist. Hierfür definieren wir die Kanäle $DtoT$ und $TtoD$. Mit dem Prozessor ist D durch die Kanäle $DtoPZ$ und $PZtoD$ verbunden, um die Kanalverbindungen zwischen Prozeß und Prozessor erzeugen und insbesondere Suspendierungen veranlassen zu können.

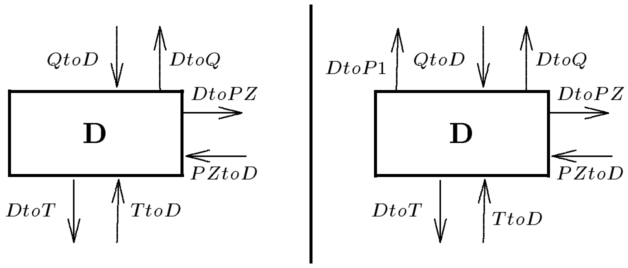


Abbildung 4.3.9: SSDs der Phasen von D

Der Dispatcher soll nicht permanent alle Prozesse des Systems *kennen*, sondern einen Prozeß nur dann zur Kenntnis nehmen, wenn dieser für die Zuteilung des Prozessors ausgewählt wurde. Somit muß der Dispatcher mit $P1$ zumindest zeitweise verbunden sein, nämlich dann, wenn die Verbindung zwischen $P1$ und PZ erzeugt werden soll. Zu diesem Zweck wurde $P1$ bereits so modelliert, daß er den Port $!DtoP1$ bei seiner Anforderung des Prozessors bekannt gibt. D erhält diesen Port von der Queue und kann somit die Kanalverbindungen zwischen PZ und $P1$ erzeugen. Abbildung 4.3.9 zeigt die beiden Phasen von D, auf deren Darstellung wir in Abbildung 4.3.1 verzichtet hatten. Wir erhalten die in Abbildung 4.3.10 vorgestellte ANDL-Spezifikation von D.

agent D
  input channels $QtoD : S_{QtoD},\ TtoD : S_{TtoD},\ PZtoD : S_{PZtoD}$
  output channels $DtoQ : S_{DtoQ},\ DtoT : S_{DtoT},\ DtoPZ : S_{DtoPZ}$
  private channels $PtoPZ : S_{PtoPZ},\ PZtoP : S_{PZtoP}$
is basic
  f_D | mit der Spezifikation von Seite 63 |
end D

Abbildung 4.3.10: ANDL-Spezifikation von D

D wird als mobile Komponente modelliert, da er die Erzeugung der Kanalverbindungen zwischen Prozeß und Prozessor initiiert und Ports empfängt. Die Menge der privaten

Kanäle ist initial festgelegt mit $pp_D = \{?!PtoPZ, ?!PZtoP\}$. In den Phasen, in denen eine Kopplung zwischen Prozessor und Prozeß besteht, gilt: $pp_D = \emptyset$.

Zur Präzisierung des Verhaltens von D und zur Festlegung der Nachrichten, die D empfängt und verschickt, geben wir folgende Erläuterungen: Das aktuell gültige Quantum wird mit $rrq \in I\!N$ (abkürzend für **r**ound-**r**obin-**q**uantum) bezeichnet. Die Nachricht *Timeout* signalisiert, daß die Zeitscheibe abgelaufen ist. Mit *Reset* wird das Messen einer Zeitscheibe gestoppt. D fordert durch Ausgabe der Nachricht *Next* über Kanal *DtoQ* den für die nächste Zuteilung des Prozessors ausgewählten Prozeß an. Empfängt D von der Queue die Nachricht *Empty*, so wird erneut ein Identifikator erfragt. Auf diese Weise realisieren wir *aktives Warten*. Mit diesen Erläuterungen ergeben sich die in Tabelle 4.3.3 aufgeführten Nachrichtenmengen.

Kanal n	Nachrichtenmengen S_n	
$QtoD$	$\{P1\} \cup \{Empty\}$	
$DtoQ$	$\{Next\}$	
$PZtoD$	$\{Term\}$	
$DtoPZ$	$\{Suspend\}$	
$TtoD$	$\{Timeout\}$	$\cup\ ?!N_{OneP}$
$DtoT$	$I\!N \cup \{Reset\}$	
$PtoPZ$	$STEP \cup \{Term\}$	
$PZtoP$	$\{Ok(Step) \mid Step \in STEP\} \cup \{Suspend\}$	

Tabelle 4.3.3: Nachrichtentypen für D

Wir spezifizieren den Dispatcher zustandsorientiert, um explizit zu dokumentieren, ob der Prozessor aktuell an einen Prozeß gebunden ist, oder nicht. Als Zustandsmenge definieren wir: $State_D = \{free, bound\}$. Für das Verhalten von D gilt:

(1) D wird durch den Erhalt von $P1$ und dem Port $!DtoP1$ über Kanal *QtoD* gestartet. D initiiert den Verbindungsaufbau mit den Kanälen *PtoPZ* und *PZtoP* zwischen $P1$ und PZ, löscht die Verbindung über $DtoP1$, sendet die Nachricht rrq über Kanal *DtoT* und geht in den Zustand *bound* über.

(2) Erhält D die Nachricht *Timeout* über Kanal *TtoD*, sendet D die Nachricht *Suspend* über Kanal *DtoPZ* und verbleibt in Zustand *bound*.

(3) Erhält D im Zustand *bound* über Kanal *PZtoD* die Ports $?!PZtoP$ und $?!PtoPZ$, sendet D über Kanal *DtoQ* die Nachricht *Next* und geht in den Zustand *free* über.

(4) Erhält D über Kanal *TtoD* die Nachricht *Timeout* und zusätzlich die Nachricht *Term* sowie die Ports $?!PZtoP$ und $?!PtoPZ$ über Kanal *PZtoD*, so sendet D die Nachricht *Next* über Kanal *DtoQ* und geht in den Zustand *free* über.

(5) Erhält D die Nachricht *Term* sowie die Ports $?!PZtoP$ und $?!PtoPZ$ über Kanal $PZtoD$ und kein $Timeout$ über Kanal $TtoD$, sendet D die Nachrichten *Reset* über Kanal $DtoT$, *Next* über Kanal $DtoQ$ und geht in den Zustand *free* über.

(6) Erhält D im Zustand *free* über Kanal $QtoD$ die Sequenz $\langle P1, !DtoP1 \rangle$, initiiert D den Verbindungsaufbau zwischen $P1$ und PZ mit den Kanälen $PtoPZ$ und $PZtoP$, sendet die Nachricht rrq über Kanal $DtoT$ und geht in den Zustand *bound* über.

(7) Erhält D im Zustand *free* über Kanal $QtoD$ die Nachricht *Empty*, so verbleibt er in Zustand *free* und sendet die Nachricht *Next* über Kanal $DtoQ$.

Diese textuelle Beschreibung wird in folgende Formalisierung umgesetzt.

Funktionsgleichungen für f_D

$$\forall s \in \prod_{n \in N_{OneP}} [S_n^*],\ rrq \in I\!N \ :\ \exists h \in State_D \ \rightarrow\ Type_D \ :$$

(1) $f_D(\{QtoD \mapsto \langle P1, !DtoP1 \rangle\} \circ s)$
$= \{DtoT \mapsto \langle rrq \rangle, DtoP1 \mapsto \langle ?PZtoP, !PtoPZ, !DtoP1 \rangle$
$DtoPZ \mapsto \langle !PZtoP, ?PtoPZ \rangle\} \circ h(bound)(s)$

(2) $h(bound)(\{TtoD \mapsto \langle Timeout \rangle, PtoPZ \mapsto \langle\rangle\} \circ s)$
$= \{DtoPZ \mapsto \langle Suspend \rangle\} \circ h(bound)(s)$

(3) $h(bound)\{PZtoD \mapsto \langle ?!PtoPZ, ?!PZtoP \rangle\} \circ s) \ = \ \{DtoQ \mapsto \langle Next \rangle\} \circ h(free)(s)$

(4) $h(bound)(\{TtoD \mapsto \langle Timeout \rangle, PZtoD \mapsto \langle Term, ?!PtoPZ, ?!PZtoP \rangle\} \circ s)$
$= \{DtoQ \mapsto \langle Next \rangle\} \circ h(free)(s)$

(5) $h(bound)(\{TtoD \mapsto \langle\rangle, PZtoD \mapsto \langle Term, ?!PtoPZ, ?!PZtoP \rangle\} \circ s)$
$= \{DtoT \mapsto \langle Reset \rangle, DtoQ \mapsto \langle Next \rangle\} \circ h(free)(s)$

(6) $h(free)(\{QtoD \mapsto \langle P1, !DtoP1 \rangle\} \circ s)$
$= \{DtoT \mapsto \langle rrq \rangle, DtoP1 \mapsto \langle ?PZtoP, !PtoPZ, !DtoP1 \rangle,$
$DtoPZ \mapsto \langle !PZtoP, ?PtoPZ \rangle\} \circ h(bound)(s)$

(7) $h(free)(\{QtoD \mapsto \langle Empty \rangle\} \circ s) \ = \ \{DtoQ \mapsto \langle Next \rangle\} \circ h(free)(s)$

Mit (2) wird beschrieben, daß die Zeitscheibe, für deren Dauer $P1$ über den Prozessor verfügen konnte, abgelaufen ist. Die Suspendierung muß veranlaßt werden, ist aber erst dann erfolgreich abgeschlossen, sobald D die Ports für die Verbindungen zwischen Prozeß und Prozessor erhalten hat. (4) beschreibt den Fall, daß die Terminierung des Prozesses mit dem Ende der Zeitscheibe zusammenfällt.

Bis auf die Gleichungen (1) und (6) sind alle Gleichungen mit dem allgemeinen Schema (2.7) auf Seite 20 für pulsgetriebene Funktionen spezifiziert. In den Gleichungen (3), (4) und (5) empfängt D sämtliche Ports zu den beiden Kanälen $PtoPZ$ sowie $PZtoP$. Diese werden in die Menge der privaten Ports pp_D eingeordnet, wobei die Einordnung in der Semantik verborgen ist. Mit den Gleichungen (1) und (6) wird spezifiziert, daß D einen Verbindungsaufbau initiiert, wobei *neue* Kanalverbindungen erzeugt werden. Die Voraussetzung $\{?!PtoPZ, ?!PZtoP\} \subseteq pp_D$ ist in dieser Phase für den Dispatcher erfüllt, zudem verfügt D jeweils über eine Schreibverbindung zu den Komponenten PZ und P1. Die beiden Gleichungen wurden entsprechend dem Schema (2.13) von Seite 26 erstellt, wobei jeweils zwei Ports versendet werden. $\langle P1, !DtoP1 \rangle$ wurde für den Platzhalter *new* eingesetzt.

4.3.6 Der Timer

Der *Timer* T ist die Basiskomponente, die den Ablauf der Zeitscheibe $rrq \in \mathbb{N}$ während der Kopplung des Prozessors mit dem aktuell zugeteilten Prozeß mißt. Da der Dispatcher für die Realisierung des Umschaltvorgangs zuständig ist, sind Timer und Dispatcher durch die Kanäle $DtoT$ und $TtoD$ verbunden, wobei T lesend auf $DtoT$ und schreibend auf $TtoD$ zugreift. Die Schnittstelle von T ändert sich während eines Systemablaufs nicht. Es gelten $pp_T = \emptyset$ und $ap_T = \{?DtoT\} \cup \{!TtoD\}$

agent T
  input channels $DtoT : S_{DtoT}$
  output channels $TtoD : S_{TtoD}$
  private channels $\emptyset$
is basic
  $\boldsymbol{f_T}$ mit der Spezifikation von Seite 65
end T

Abbildung 4.3.11: ANDL-Spezifikation von T

Mit der Spezifikation des Dispatchers wurde bereits erklärt, daß der Timer die Nachrichten rrq und *Reset* empfängt und die Nachricht *Timeout* verschickt. Die Mengen S_{DtoT} und S_{TtoD} wurden bereits in Tabelle 4.3.3 festgelegt. Das SSD für den Timer können wir als Ausschnitt aus dem in Abbildung 4.3.1 gezeigten SSD entnehmen und erhalten die in Abbildung 4.3.11 gezeigte ANDL-Spezifikation. Für das Verhalten von T legen wir fest, daß rrq gespeichert und mit jedem Zeitintervall, das vergeht, um 1 heruntergezählt wird. Sobald die Zeitscheibe abgelaufen ist, meldet T dies dem Dispatcher. Beendet ein Prozeß seine Berechnung, bevor das Quantum abgelaufen ist, wird der Timer gestoppt.

Das Verhalten des Timers wird beschrieben durch:

(1) Empfängt T über Kanal $DtoT$ die Nachricht $rrq \in \mathbb{N}$, wird rrq gespeichert.

(2) Erhält T im Zustand $m \in \mathbb{N}$ über Kanal $DtoT$ eine von *Reset* verschiedene Nachricht, wird m um 1 heruntergezählt.

(3) Schritt (2) wird wiederholt, bis die Zeitscheibe abgelaufen ist. Dies ist der Fall, wenn der interne Zähler von T den Wert 1 erreicht hat und das nächste Zeitintervall vergangen ist. Dann sendet T die Nachricht *Timeout* über Kanal *TtoD*.

(4) Erhält T im Zustand $m \in I\!N$ die Nachricht *Reset* über Kanal *DtoT*, so wird das Messen der Zeitscheibe gestoppt.

Funktionsgleichungen für f_T
$\forall s \in \prod_{n \in N_{OneP}} [S_n^*],\ rrq, m \in I\!N,\ t \in S_{TtoD}^* \setminus \{\langle Reset \rangle\} \ :\ \exists h \in I\!N \ \rightarrow \ Type_T \ :$
(1) $f_T(\{DtoT \mapsto \langle rrq \rangle\} \circ s) \ = \ h(rrq)(s)$
Für $m > 1$
(2) $h(m)(\{DtoT \mapsto t\} \circ s) \ = \ \{TtoD \mapsto \langle\rangle\} \circ h(m-1)(s)$
(3) $h(1)(\{DtoT \mapsto t\} \circ s) \ = \ \{TtoD \mapsto \langle Timeout \rangle\} \circ f_T(s)$
(4a) $h(m)(\{DtoT \mapsto \langle Reset \rangle\} \circ s) \ = \ \{TtoD \mapsto \langle\rangle\} \circ f_T(s)$
(4b) $f_T(\{DtoT \mapsto \langle Reset \rangle\} \circ s) \ = \ \{TtoD \mapsto \langle\rangle\} \circ f_T(s)$

Alle hier gezeigten Gleichungen ergeben sich direkt mit dem allgemeinen Schema (2.7) von Seite 20 für pulsgetriebene Funktionen. Die explizite Nennung der Ausgabe von $\langle\rangle$ in den Gleichungen (2),(4a) und (4b) ist zwar nicht nötig, da es durch die von uns festgelegte Konvention (2.8) von Seite 20 abgedeckt ist. Wir haben diese Ausgabe hier jedoch zur Verdeutlichung des Voranschreitens der Zeit explizit aufgeführt.

4.3.7 Der Warteraum

Die Komponente Q bildet die Zwischenstelle zwischen Prozeß und Dispatcher. Prozesse tragen sich in diesen Warteraum ein, wenn sie den Prozessor anfordern. Q ist mit $P1$ durch den Kanal $P1toQ$, auf den der Prozeß schreibend zugreift, verbunden. Zwischen Q und dem Dispatcher muß sowohl eine Lese- als auch eine Schreibverbindung bestehen, da der Dispatcher die Prozeßidentifikatoren *aktiv* anfordert. Falls Identifikatoren abgespeichert sind, werden sie auf Anforderung an den Dispatcher gesendet. Für diese Verbindungen sind die Kanäle *DtoQ* und *QtoD*, auf die Q lesend bzw. schreibend zugreift, definiert. Das SSD für die Queue kann aus dem in Abbildung 4.3.1 gezeigten SSD entnommen werden.

Q empfängt über Kanal $P1toQ$ den Identifikator $P1$ und den Port $!DtoP1$. Über $DtoP1$ kann Q den ausgewählten Prozeß mit dem Dispatcher verbinden. Q gibt den Port $!DtoP1$ an D weiter, sobald der Prozeß $P1$ zur Zuteilung des Prozessors ausgewählt wird. Von D empfängt Q mit der Nachricht *Next* Anfragen nach einem nächsten Identifikator. Ist der Warteraum leer, so wird dies dem Dispatcher mit der Nachricht *Empty* gemeldet. Die den

Kanälen *DtoQ*, *QtoD* und *P1toQ* zugeordneten Nachrichtenmengen wurden bereits in den Tabellen 4.3.3 auf Seite 62 und 4.3.1 auf Seite 55 festgelegt und können dort nachgelesen werden. Für Q gilt die ANDL-Spezifikation aus Abbildung 4.3.12.

```
agent Q
  input channels     P1toQ : S_P1toQ, DtoQ : S_DtoQ
  output channels    QtoD : S_QtoD
  private channels   ∅
is basic
    f_Q  [mit der Spezifikation von Seite 67]
end Q
```

Abbildung 4.3.12: ANDL-Spezifikation von Q

Queue wird durch den Erhalt des Identifikators P1 gestartet und speichert diesen ab. Die Speicherung erfolgt gemäß der FIFO-Strategie[1]. Um die abgespeicherten Identifikatoren explizit zu beschreiben, modellieren wir Q zustandsorientiert. Als Schaltstelle zwischen anforderndem Prozeß und dem Dispatcher muß Q jeden empfangenen Identifikator speichern. Vom Dispatcher werden die Identifikatoren angefordert. Enthält der Warteraum einen Identifikator, so wird das erste Element des internen Speichers an den Dispatcher geschickt. In der im folgenden angegebenen detaillierten Verhaltensbeschreibung bezeichnen wir mit p die gespeicherte Liste von Prozeßidentifikatoren:

(1) Erhält Q über Kanal *P1toQ* den ersten Identifikator $P1$ und den Port $!DtoP1$, so werden beide über Kanal *QtoD* gesendet.

(2) Empfängt Q über Kanal *DtoQ* die Nachricht *Next* und gilt $p \neq \langle\rangle$, sendet Q die Nachricht $ft.p$ sowie den Port $DtoP1$ über Kanal *QtoD*. Erhält Q zudem über Kanal *P1toQ* den Identifikator $P1$, so wird dieser abgespeichert.

(3) Erhält Q über Kanal *DtoQ* die Nachricht *Next* und gilt $p = \langle\rangle$, sendet Q die Nachricht *Empty* über Kanal *QtoD* und verbleibt im Zustand $\langle\rangle$.

(4) Erhält Q sowohl über Kanal *P1toQ* den Identifikator $P1$ und den Port $!DtoP1$ als auch die Nachricht *Next* über Kanal *DtoQ* und gilt $p = \langle\rangle$, werden $P1$ und der Port $!DtoP1$ über Kanal *QtoD* gesendet.

(5) Erhält Q über Kanal *P1toQ* den Identifikator $P1$ sowie den Port $!DtoP1$ und keine Anfrage *Next* über Kanal *DtoQ*, wird $P1$ abgespeichert.

Alle im folgenden gezeigten Gleichungen ergeben sich direkt mit dem allgemeinen Schema (2.7) von Seite 20 für pulsgetriebene Funktionen. Zusätzlich ergeben sich die Gleichungen (1), (2) und (4) gemäß dem Schema (2.14) von Seite 27.

[1]FIFO ist die Abkürzung von **F**irst-**I**n-**F**irst-**O**ut. Hierbei muß das Element, das zuerst abgespeichert wurde, auch zuerst wieder ausgegeben werden.

Funktionsgleichungen für f_Q
$\forall s \in \prod_{n \in N_{OneP}} [S_n^*],\ p \in \{P1\}^*\ :\ \exists h \in \{P1\}^* \ \to\ Type_Q\ :$
(1) $f_Q(\{P1toQ \mapsto \langle P1, !DtoP1\rangle\} \circ s) \ = \ \{QtoD \mapsto \langle P1, !DtoP1\rangle\} \circ h(\langle\rangle)(s)$
Für $t \in \{\langle P1, !DtoP1\rangle, \langle\rangle\}$ und $p \neq \langle\rangle$:
(2) $h(p)(\{P1toQ \mapsto t, DtoQ \mapsto \langle Next\rangle\} \circ s) \ = \ \{QtoD \mapsto \langle ft.p, !DtoP1\rangle\} \circ h(rt.p \circ (\{P1\} \copyright t))(s)$
(3) $h(\langle\rangle)(\{P1toQ \mapsto \langle\rangle, DtoQ \mapsto \langle Next\rangle\} \circ s) \ = \ \{QtoD \mapsto \langle Empty\rangle\} \circ h(\langle\rangle)(s)$
(4) $h(\langle\rangle)(\{P1toQ \mapsto \langle P1, !DtoP1\rangle, DtoQ \mapsto \langle Next\rangle\} \circ s) \ = \ \{QtoD \mapsto \langle P1, !DtoP1\rangle\} \circ h(\langle\rangle)(s)$
(5) $h(p)(\{P1toQ \mapsto \langle P1, !DtoP1\rangle, DtoQ \mapsto \langle\rangle\} \circ s) \ = \ \{QtoD \mapsto \langle\rangle\} \circ h(p \circ P1)(s)$

4.4 Dispatching für zwei Prozesse

Das in Abschnitt 4.3 beschriebene System stellt eine stark vereinfachte Konstellation im Rahmen der Prozessorverwaltung dar. Im allgemeinen bewirbt sich eine große Anzahl von Prozessen um den Prozessor als exklusiv benutzbares Betriebsmittel. Bei der systematischen Erarbeitung der formalen Modellierung des Systems zur Prozessorverwaltung bestand der erste Schritt darin, zunächst ein *Kernsystem* zu spezifizieren. Dieses System weist bereits einige für die später zu lösende Aufgabe charakteristische Merkmale auf und stellt die Basis für die von uns entwickelte Systemspezifikation dar.

Das in Abschnitt 4.3 vorgestellte System wurde im Hinblick auf die Verwaltung des Prozessors für mehrere Prozesse spezifiziert. Dieser Schritt ist die Basis für unsere Methode zur Entwicklung eines vollständigen Spezifikation für ein System zur Verwaltung von Betriebsmitteln. Mit dieser Vorgehensweise orientiert sich die Weiterentwicklung der Formalisierung an der steigenden Komplexität der Aufgabenstellung. Aufgrund der vorbereitenden Maßnahmen in den *einfachen* Spezifikationen entstehen deren Erweiterungen durch Anpassungen und Vervollständigungen. Die formalen Modellierungen der Systeme mit wachsender Komplexität sind schrittweise nachvollziehbar und erscheinen dadurch leichter verständlich, als die direkte Modellierung des Systems in seinem vollen Umfang. Bei dem nun behandelten erweiterten Systemaufbau mit zwei konkurrierenden Prozessen bauen wir auf der im vorherigen Abschnitt erstellten Spezifikation auf. Die Aufgabenstellung lautet allgemein:

> Für ein Einprozessorsystem werden die Zuteilung und Freigabe des exklusiv benutzbaren Betriebsmittels „Prozessor“ für zwei Prozesse modelliert.

Betrachten wir ausgehend von dem in Abschnitt 4.3.1 gezeigten Systemaufbau die hier auftretenden Komponenten:

Prozessor: Im System ist weiterhin *ein Prozessor* enthalten, der zu jedem Zeitpunkt von einem Prozeß genutzt werden kann. Für den Prozessor ist es irrelevant, an welchen Prozeß er aktuell gebunden ist, er führt die Berechnungsschritte aus, die an ihn gesendet werden. Er leitet Suspendierungs- und Terminierungsmeldungen weiter. Da die Nachrichten, die der Prozessor empfängt, unabhängig von der Anzahl der im System existierenden Prozesse sind, müssen an der Modellierung des Prozessors keine Änderungen vorgenommen werden.

Prozesse: Im System sind nun *zwei Prozesse* enthalten. Ein Prozeß benötigt den Prozessor zur Durchführung der ihn charakterisierenden Berechnung. Da wir Prozesse betrachten, die unabhängig voneinander arbeiten, wird der zweite Prozeß entsprechend zu der in Abschnitt 4.3.3 gegebenen Spezifikation des Prozesses $P1$ modelliert. Die Bezeichner müssen entsprechend gewählt werden, da sie für Schnittstellendefinitionen in Focus eindeutig sein müssen. Die Spezifikation von $P1$ wird vollständig übernommen.

Timer: Der *Timer* ist dafür zuständig, den Ablauf einer Zeitscheibe zu messen und deren Ende dem Dispatcher zu melden. Er ist ausschließlich mit dem Dispatcher verbunden und wird immer mit Zeitmarke rrq gestartet, die vom aktuell in Ausführung befindlichen Prozeß unabhängig ist. Die Spezifikation des Timers aus Abschnitt 4.3.6 kann somit ohne Änderungen wiederverwendet werden.

Dispatcher: Der *Dispatcher* ist dafür zuständig, die Verbindung zwischen dem aktuell ausgewählten Prozeß und dem Prozessor herzustellen. Zur Erzeugung dieser Verbindung muß er mit dem ausgewählten Prozeß verbunden sein. Den entsprechenden Port erhält der Dispatcher zusammen mit dem Prozeßidentifikator. Die Modellierung muß entsprechend angepaßt werden.

Queue: In Abschnitt 4.3.7 haben wir erläutert, daß die *Queue* die Aufgaben eines Schedulers übernimmt. Da sich in dem nun betrachteten System zwei Prozesse mit gleicher Priorität um den Prozessor bewerben, wird die Queue so spezifiziert, daß keine Anmeldung eines Prozesses verloren geht, kein Prozeß bevorzugt wird und die *richtigen* Ports weitergeleitet werden. Die Modellierung der Queue wird erweitert.

Auf welche Weise die in den Abschnitten 4.3.3, 4.3.5 und 4.3.7 vorgestellten Spezifikationen für Prozeß P2, Dispatcher und Queue im Detail erweitert bzw. inwieweit sie wiederverwendet werden, zeigen die folgenden Abschnitte.

4.4.1 Ein System zur Prozessorverwaltung mit zwei Prozessen

Betrachten wir den erweiterten Systemaufbau relativ zu dem System aus Abschnitt 4.3. Das System besteht auch weiterhin aus einem *Prozessor*, einem *Dispatcher*, einem *Timer* und einer *Queue*. Als Erweiterung sind im System zwei Prozesse $P1$ und $P2$ enthalten. In allen folgenden Spezifikationen werden wir die wesentlichen Erweiterungen relativ zu den vorher gezeigten Modellierungen **hervorheben** und erläutern.

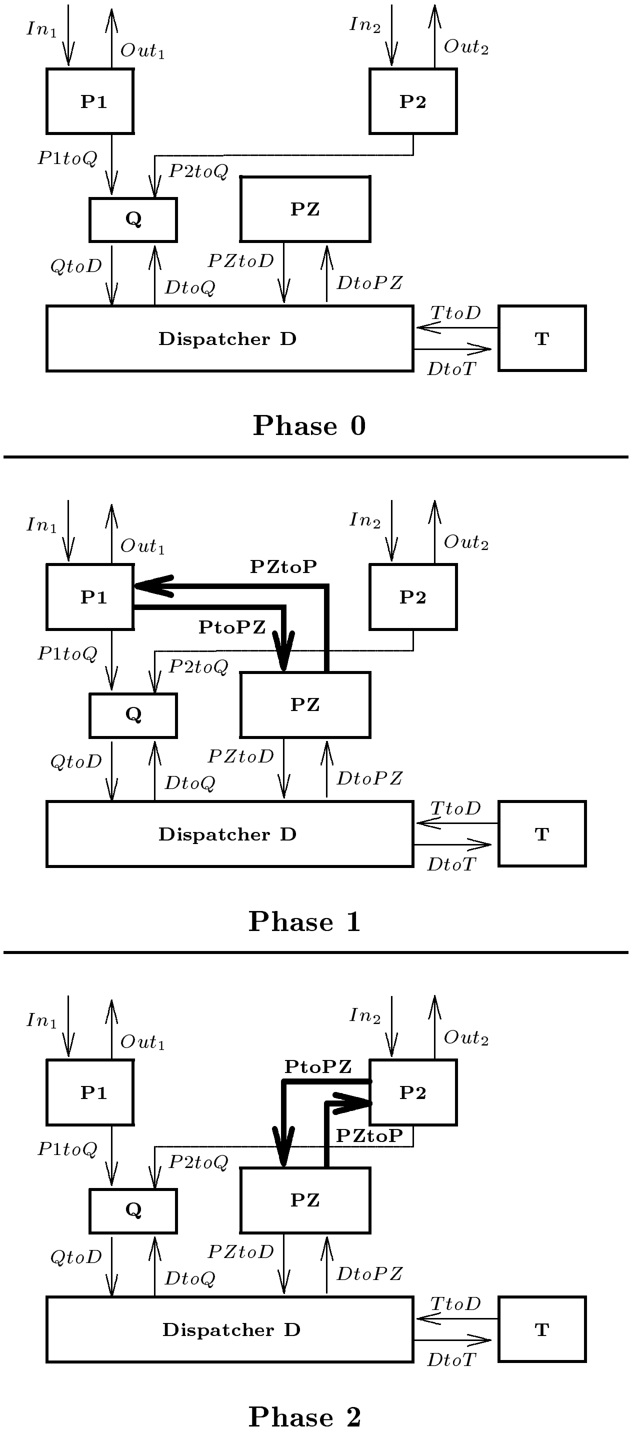


Abbildung 4.4.1: Prozessorzuteilung im System `TwoP` mit zwei Prozessen

Die beiden Prozesse bilden mit den Kanälen In_1, In_2, Out_1 und Out_2 die Schnittstelle des Systems zur Umgebung. Der Dispatcher ist mit der Queue, dem Timer und dem Prozessor verbunden. Entsprechend zum System mit einem Prozeß ist der ausgewählte Prozeß kurzfristig mit dem Dispatcher verbunden, um an den Prozessor gekoppelt zu werden. Jeder Prozeß ist mit der Queue verbunden, um den Prozessor anfordern zu können. Das System wird als mobiles, dynamisches Netz modelliert, in dem es im wesentlichen 3 Phasen gibt: In *Phase 0* ist der Prozessor weder P1 noch P2 zugeteilt, in *Phase 1* ist P1 und in *Phase 2* ist P2 an den Prozessor gebunden. Diese Kopplung modellieren wir entsprechend zu dem in Abschnitt 4.3.1 gezeigten System durch das Erzeugen und Löschen der Lese- bzw. Schreibverbindungen mit den Kanälen $PtoPZ$ und $PZtoP$. Mit diesen Erläuterungen erhalten wir die in Abbildung 4.4.1 gezeigte Folge von SSDs, die die oben genannten Phasen veranschaulichen. Die Umsetzung des SSD zur initialen Vernetzung der Komponenten in diesem System liefert die in Abbildung 4.4.2 gezeigte ANDL-Spezifikation für `TwoP`.

agent `TwoP`
  input channels $In_1 : S_{In_1}, \boldsymbol{In_2 : S_{In_2}}$
  output channels $Out_1 : S_{Out_1}, \boldsymbol{Out_2 : S_{Out_2}}$
is network

$$
\begin{array}{lcl}
\ll Out_1, P1toQ \gg & = & \texttt{P1} \ll In_1 \gg \; ; \\
\boldsymbol{\ll Out_2, P2toQ \gg} & \boldsymbol{=} & \boldsymbol{P2 \ll In_2 \gg} \; ; \\
\ll QtoD \gg & = & \texttt{Q} \ll P1toQ, \boldsymbol{P2toQ}, DtoQ \gg \; ; \\
\ll DtoQ, DtoT, DtoPZ \gg & = & \texttt{D} \ll QtoD, TtoD, PZtoD \gg \; ; \\
\ll PZtoD \gg & = & \texttt{PZ} \ll DtoPZ \gg \; ; \\
\ll TtoD \gg & = & \texttt{T} \ll DtoT \gg \; ;
\end{array}
$$

end `TwoP`

Abbildung 4.4.2: ANDL-Spezifikation für das System `TwoP` mit zwei Prozessen

Im folgenden betrachten wir die Komponenten, deren Spezifikationen gemäß dem oben kurz beschriebenen Verhalten angepaßt bzw. erweitert werden müssen. Die für das betrachtete Netz gültige Menge aller Kanalbezeichner bezeichnen wir mit N_{TwoP}.

4.4.2 Der zweite Prozeß P2

Gemäß der bereits gegebenen Beschreibung des Verhaltens eines Prozesses zeigt sich, daß wir zur Spezifikation von $P2$ nur die Schnittstelle neu definieren müssen. $P2$ wird als mobile Komponente modelliert, die zwei Phasen durchläuft. $P2$ ist an den Prozessor gebunden, oder dies ist nicht der Fall. $P2$ wird mit $pp_{P2} = \{?!DtoP2\}$ initialisiert, und die Phasen sind charakterisiert durch ihre aktuelle Schnittstelle

$$ap_{P2} = \{?In_2, !Out_2, !P2toQ, ?DtoP2\} \qquad \text{bzw.}$$

$$ap_{P2} = \{?In_2, !Out_2, !P2toQ, ?PZtoP, !PtoPZ\}$$

Aus dem SSD für das gesamte System von Abbildung 4.4.1 entnehmen wir die Schnittstelle zu $P2$ und erhalten die mit Abbildung 4.4.3 gezeigte ANDL-Spezifikation.

agent P2
  input channels $In_2 : S_{In_2}$
  output channels $Out_2 : S_{Out_2},\ P2toQ : S_{P2toQ}$
  private channels $\emptyset$
is basic
  f_{P2} mit der Spezifikation von Seite 57 und angepaßter Schnittstelle
end P2

Abbildung 4.4.3: ANDL-Spezifikation von P2

Zur Festlegung der zugeordneten Nachrichtenmengen verweisen wir auf Abschnitt 4.3.3 und die Tabellen 4.4.1 und 4.4.2. Insgesamt wird die Spezifikation des Verhaltens von $P2$ vollständig gemäß der Spezifikation von P1 in Abschnitt 4.3.3 übernommen, wobei in der informellen Beschreibung und in allen Funktionsgleichungen $P1$ durch $P2$ ersetzt wird.

4.4.3 Der erweiterte Dispatcher

Wie bereits zu Beginn von Abschnitt 4.4 erläutert, sind an der Spezifikation des Dispatchers D nur geringfügige Änderungen vorzunehmen. Entsprechend zur ersten Spezifikation in Abschnitt 4.3.5 wird D als mobile Komponente modelliert, da D in der Lage sein muß, die Erzeugung von Kanalverbindungen zu initiieren. D erhält zusätzlich zum Identifikator des ausgewählten Prozesses einen Port, der ihm das Schreibrecht an einer Kanalverbindung zum Prozeß zuteilt. Ausgehend vom SSD für das gesamte System mit zwei Prozessen erhalten wir die in Abbildung 4.4.4 gezeigte ANDL-Spezifikation für den erweiterten Dispatcher D, an der relativ zu Abbildung 4.3.10 auf Seite 61 keine Änderung erkennbar ist. Die Erweiterung des Dispatchers wird erst anhand der Verhaltensspezifikation deutlich.

agent D
  input channels $QtoD : S_{QtoD},\ TtoD : S_{TtoD},\ PZtoD : S_{PZtoD}$
  output channels $DtoQ : S_{DtoQ},\ DtoT : S_{DtoT}, DtoPZ : S_{DtoPZ}$
  private channels $PtoPZ : S_{PtoPZ},\ PZtoP : S_{PZtoP}$
is basic
  f_{D} mit der Spezifikation von Seite 63 und Seite 72
end D

Abbildung 4.4.4: ANDL-Spezifikation von D im System mit zwei Prozessen

Die den Kanälen zugeordneten Nachrichtenmengen können wir fast vollständig aus den in Abschnitt 4.3 gezeigten Spezifikationen entnehmen. Zusätzlich kommen die in Tabelle

4.4.1 aufgeführten Festlegungen hinzu. Insgesamt ist zu berücksichtigen, daß im hier beschriebenen System allen Kanälen die Ports aus der Menge $?!N_{TwoP}$ zugeordnet werden.

Kanal n	Nachrichtenmengen S_n
$QtoD$	$\{P1, \boldsymbol{P2}\} \cup \{Empty\} \cup ?!N_{\boldsymbol{TwoP}}$
$\boldsymbol{DtoP2}$	$\emptyset \cup ?!N_{\boldsymbol{TwoP}}$

Tabelle 4.4.1: Nachrichtentypen für D

Da wir die Kanalbezeichner für die Verbindungen zwischen Prozeß und Prozessor unabhängig von den im System existierenden Prozessen gewählt haben, müssen wir bei der Spezifikation des Verhaltens von D zusätzlich nur den Fall betrachten, in dem D den Identifikator $\boldsymbol{P2}$ und den Port $\boldsymbol{!DtoP2}$ von der Queue empfängt. Dieses veränderte Verhalten des Dispatchers hat Auswirkungen auf das auf Seite 63 durch die Punkte (1) und (6) beschriebene Verhalten, das wir wie folgt anpassen:

(1) D wird durch die Sequenz $\langle P1, !DtoP1 \rangle$ **oder** $\langle \boldsymbol{P2}, \boldsymbol{!DtoP2} \rangle$ über Kanal $QtoD$ gestartet. Er initiiert den Verbindungsaufbau mit den Kanälen $PtoPZ$ und $PZtoP$ zwischen $P1$ **bzw.** $\boldsymbol{P2}$ und PZ, sendet die Nachricht rrq über Kanal $DtoT$ und geht in den Zustand *bound* über.

(6) Erhält D im Zustand *free* die Nachricht $P1$ **bzw.** $\boldsymbol{P2}$ mit dem Port $!DtoP1$ **bzw.** $\boldsymbol{!DtoP2}$, so initiiert D den Verbindungsaufbau zwischen $P1$ und PZ **bzw.** $\boldsymbol{P2}$ und PZ über die Kanäle $DtoP1$ bzw. $\boldsymbol{DtoP2}$ und $DtoPZ$, sendet die Nachricht rrq über Kanal $DtoT$ und geht in den Zustand *bound* über.

Dieses erweiterte Verhalten hat zur Folge, daß die Gleichungen (1) und (6) der formalen Spezifikation des Dispatchers von Seite 63 entsprechend angepaßt werden müssen. Diese ergeben sich durch Umsetzung der textuellen Beschreibung nach Schema (2.13) von Seite 26 und entsprechend zu (6) bzw. (1) von Seite 63 mit $\langle P2, !DtoP2 \rangle$ und $DtoP2$.

Erweiterung der Spezifikation für f_D von Seite 63

Für $\boldsymbol{i \in \{1, 2\}}$:

(1) $f_D(\{QtoD \mapsto \langle \boldsymbol{Pi}, \boldsymbol{!DtoPi} \rangle\} \circ s)$
$$= \{DtoT \mapsto \langle rrq \rangle, \boldsymbol{DtoPi} \mapsto \langle ?PZtoP, !PtoPZ, \boldsymbol{!DtoPi} \rangle, DtoPZ \mapsto \langle !PZtoP, ?PtoPZ \rangle\} \circ h(bound)(s)$$

(6) $h(free)(\{QtoD \mapsto \langle \boldsymbol{Pi}, \boldsymbol{!DtoPi} \rangle\} \circ s)$
$$= \{DtoT \mapsto \langle rrq \rangle, \boldsymbol{DtoPi} \mapsto \langle ?PZtoP, !PtoPZ, \boldsymbol{!DtoPi} \rangle, DtoPZ \mapsto \langle !PZtoP, ?PtoPZ \rangle\} \circ h(bound)(s)$$

4.4.4 Der erweiterte Warteraum

Zu Beginn von Abschnitt 4.4 wurde kurz beschrieben, wie die Spezifikation von Q zu erweitern ist. Das veränderte Verhalten ist dadurch charakterisiert, daß sich *zwei* Prozesse $P1$ und $P2$ gleicher Priorität um den *Prozessor* bewerben. Am SSD zu `TwoP` von Abbildung 4.4.1 wird ersichtlich, daß sowohl $P1$ als auch $P2$ durch einen entsprechenden Kanal mit Q verbunden sind. Für die Nachrichtentypen zur Spezifikation von Q gelten die in Tabelle 4.4.2 gezeigten Erweiterungen relativ zur Spezifikation von Q aus Abschnitt 4.3.7.

Kanal n	Nachrichtenmengen S_n	
$P1toQ$	$\{P1\}$	
$\boldsymbol{P2toQ}$	$\boldsymbol{\{P2\}}$	$\cup\ ?!N_{\boldsymbol{TwoP}}$
$QtoD$	$\{Empty\} \cup \{P1, \boldsymbol{P2}\}$	

Tabelle 4.4.2: Nachrichtentypen für Q

Mit den Informationen aus Tabelle 4.4.2 und ausgehend von dem die Queue betreffenden Ausschnitt des SSD von Abbildung 4.4.1 zeigt Abbildung 4.4.5 die ANDL-Spezifikation für die erweiterte Queue.

agent Q
input channels $P1toQ : S_{P1toQ}, \boldsymbol{P2toQ : S_{P2toQ}}, DtoQ : S_{DtoQ}$
output channels $QtoD : S_{QtoD}$
private channels $\emptyset$
is basic
f_{Q} $\boxed{\text{mit der Spezifikation von Seite 76}}$
end Q

Abbildung 4.4.5: ANDL-Spezifikation von `Q` im System `TwoP`

Die Spezifikation des Verhaltens der Queue muß so erweitert werden, daß *jede* Anforderung des Prozessors durch einen Prozeß registriert und kein Prozeß bevorzugt wird. Dies erreichen wir, indem alle in einem Zeitintervall an den Eingabekanälen $P1toQ$ und $P2toQ$ anstehenden Anforderungen registriert werden. Zusätzlich werden diese in abwechselnder Reihenfolge gemäß der FIFO-Strategie abgespeichert. Q wird gestartet, indem zuerst die Anforderung von Kanal $P1toQ$ und danach die von Kanal $P2toQ$ gespeichert wird. Die Spezifikation von Q wird neu erstellt; wir orientieren uns jedoch an der in Abschnitt 4.3.7 vorgestellten Spezifikation. Insbesondere muß die Einhaltung der *richtigen* Reihenfolge zusätzlich beschrieben werden. In der folgenden textuellen Verhaltensbeschreibung bezeichnen wir mit $p \in \{P1, P2\}^*$ den internen Speicher von Q.

(1) Q wird gestartet, wenn entweder beide oder nur ein Prozeß den Prozessor anfordern. Erhält Q sowohl $\langle P1, !DtoP1 \rangle$ über Kanal $P1toQ$ als auch $\langle P2, !DtoP2 \rangle$ über Kanal $P2toQ$, werden $\langle P1, !DtoP1 \rangle$ über Kanal $QtoD$ gesendet und $P2$ gespeichert. Erhält Q nur über Kanal $P1toQ$ oder $P2toQ$ eine Anforderung, wird diese über Kanal $QtoD$ gesendet. Es wird registriert, daß im nächsten Zeitintervall die über Kanal $P2toQ$ eintreffende Anforderung zuerst gespeichert wird.

(2) Empfängt Q über Kanal $DtoQ$ die Anfrage *Next* und gilt $p \neq \langle\rangle$, sendet Q die Sequenz $\langle ft.p, !Dto(ft.p) \rangle$ über Kanal $QtoD$. Erhält Q zudem über die Kanäle $P1toQ$ bzw. $P2toQ$ die Identifikatoren $P1$ bzw. $P2$ und die Ports $!DtoP1$ bzw. $!DtoP2$, so werden diese gemäß der aktuell gültigen Reihenfolge abgespeichert.

(3) Erhält Q über Kanal $DtoQ$ die Anfrage *Next*, liegen keine weiteren Eingaben an den Kanälen $P1toQ$ bzw. $P2toQ$ an und gilt $p = \langle\rangle$, so sendet Q die Nachricht *Empty* über Kanal $QtoD$.

(4) Es gilt $p = \langle\rangle$.

(a) Erhält Q die Nachricht *Next* über Kanal $DtoQ$ und eine Anforderung des Prozessors über den Kanal $PitoQ$, der der aktuell gültigen Reihenfolge entspricht, so wird diese Anforderung über Kanal $QtoD$ an den Dispatcher gesendet. Falls vom zweiten Prozeß ebenfalls eine Anforderung des Prozessors vorliegt, wird diese abgespeichert.

(b) Q erhält die Nachricht *Next* über Kanal $DtoQ$, und am Kanal $PitoQ$, der der aktuell gültigen Reihenfolge entspricht, liegt keine Anforderung des Prozessors als Eingabe vor. Wenn hierbei die Anmeldung des zweiten Prozesses vorliegt, wird diese über Kanal $QtoD$ gesendet.

(5) Erhält Q über die Kanäle $P1toQ$ bzw. $P2toQ$ die Anforderungen von $P1$ bzw. $P2$ und über Kanal $DtoQ$ eine von *Next* verschiedene Nachricht, so werden die Identifikatoren gemäß der aktuell gültigen Reihenfolge abgespeichert.

Die bereits erwähnte *richtige* Reihenfolge modellieren wir, indem die an den Kanälen $P1toQ$ und $P2toQ$ in jedem Zeitintervall eintreffenden Prozeßidentifikatoren in alternierender Reihenfolge abgespeichert werden, wobei wir mit Kanal $P1toQ$ beginnen. In der Spezifikation der Queue führen wir hierzu einen Zustandsparameter zur Festlegung der Reihenfolge ein, in der die eintreffenden Identifikatoren im internen Speicher abgespeichert werden. Dieser Zustandsparameter wird mit „2“ initialisiert. Mit jedem Verstreichen eines Zeitintervalls wird er auf $j = (l \texttt{ mod } 2) + 1$ gesetzt. Damit nimmt der Zustandsparameter alternierend die Werte „1“ und „2“ an.

Die Umsetzung der Anforderungen (1) bis (5) ergeben sich mit dem allgemeinen Schema (2.7) von Seite 20 für pulsgetriebene Funktionen. Da jedoch zusätzliche Variablen verwendet und einige Fallunterscheidungen explizit spezifiziert werden, geben wir weitere Erläuterungen zu den formalen Spezifikationen an.

zu (1) Es wird unterschieden, ob von beiden oder nur von einem Prozeß die Anforderung nach dem Prozessor ansteht. (1) könnte auch spezifiziert werden durch

$$\begin{aligned} & f_Q(\{P1toQ \mapsto \langle P1, !DtoP1 \rangle, P2toQ \mapsto in_2\} \circ s) \\ & \qquad = \{QtoD \mapsto \langle P1, !DtoP1 \rangle\} \circ h(2, \{P2\} \copyright in_2)(s) \\ & f_Q(\{P1toQ \mapsto \langle\rangle, P2toQ \mapsto \langle P2, !DtoP2 \rangle\} \circ s) \\ & \qquad = \{QtoD \mapsto \langle P2, !DtoP2 \rangle\} \circ h(2, \langle\rangle)(s) \end{aligned}$$

Zur Einhaltung der *richtigen* Reihenfolge muß h mit „2" initialisiert werden.

zu (2) Gilt $p \neq \langle\rangle$, wird infolge der Nachricht *Next* das erste Element $ft.p$ des internen Speichers ausgegeben. Die eintreffenden Prozeßidentifikatoren werden gemäß dem aktuellen Wert von l gespeichert. D.h. bei $l = \mathbf{1}$ wird $\{P1, P2\} \copyright (in_{\mathbf{1}} \circ in_2)$ und bei $l = \mathbf{2}$ wird $\{P1, P2\} \copyright (in_{\mathbf{2}} \circ in_1)$ an den Speicherinhalt angehängt. Der Zustandsparameter wird mit $j = (l \ \mathtt{mod}\ 2) + 1$ aktualisiert, d.h. bei $l = 1$ mit $j = 2$ und bei $l = 2$ mit $j = 1$. Mit $in_i \in \{\langle Pi, !DtoPi \rangle, \langle\rangle\}$ werden die Fälle, in denen aktuell kein Prozeß einen Prozessor anfordert, ebenfalls behandelt.

zu (3) Die Gleichung (3) ergibt sich direkt aus der Beschreibung von Seite 74. Auch hier muß der Parameter l entsprechend aktualisiert werden.

zu (5) Diese Gleichung gewährleistet, daß immer *alle* eintreffenden Identifikatoren gemäß der aktuell gültigen Reihenfolge abgespeichert werden; keine Prozessoranforderungen geht verloren. Dies muß vor allem dann gelten, wenn vom Dispatcher aktuell keine Anfrage *Next* vorliegt. Der Parameter l muß mit $j = (l \ \mathtt{mod}\ 2) + 1$ aktualisiert werden.

zu (4) Im aktuellen Zeitintervall seien folgende Voraussetzungen erfüllt:

1. Es liegen keine Prozessoranforderungen vor,
2. der Dispatcher fordert mit *Next* den nächsten Identifikator an und
3. mindestens einer der beiden Prozesse fordert den Prozessor an.

Wir müssen zwei Konstellationen separat betrachten:

(a) Der Prozeß, dessen Anforderung gemäß der aktuell gültige Reihenfolge zuerst behandelt wird, fordert den Prozessor an: $\{PltoQ \mapsto \langle Pl, !DtoPl \rangle\}$. $\langle Pl, !DtoPl \rangle$ wird an D weitergeleitet und eine möglicherweise vorliegende Anforderung durch den anderen Prozeß wird abgespeichert.

(b) Von dem Prozeß, dessen Anforderungen durch die aktuell gültige Reihenfolge zuerst behandelt wird, liegt keine Anforderung vor. Da jedoch vom anderen Prozeß jedoch eine Anforderung vorliegt, wird diese an den Dispatcher gesendet.

In beiden Fällen wird der Zustandsparameter l durch $j = (l \ \mathtt{mod}\ 2) + 1$ aktualisiert.

Wir erhalten die folgende Umsetzung des textuell beschriebenen Verhaltens.

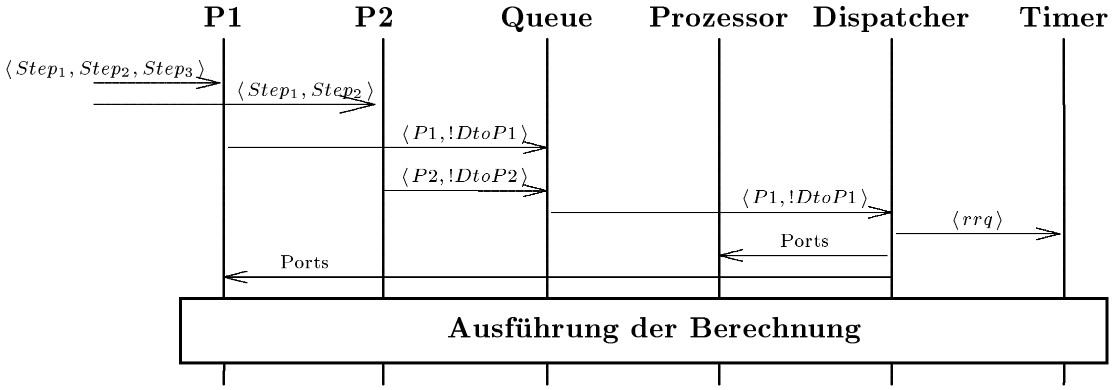
Funktionsgleichungen für f_Q
$\forall s \in \prod_{n \in N_{TwoP}} [S_n^*],\ rrq \in I\!N,\ k,l,j \in \{1,2\},\ p,q,r \in \{P1,P2\}^*,\ in_k \in \{\langle Pk, !DtoPk\rangle, \langle\rangle\}\ :$ $\exists h \in (\{1,2\} \times \{P1,P2\}^*) \rightarrow Type_Q\ :$
Für $(in_1 = \langle P1, !DtoP1\rangle\ \vee\ in_2 = \langle P2, !DtoP2\rangle)\ :$ (1) $f_Q(\{P1toQ \mapsto in_1, P2toQ \mapsto in_2\} \circ s)\ =\ \{QtoD \mapsto \langle ft.r, !Dto(ft.r)\rangle\} \circ h(2, rt.r)(s)$ wobei $r\ =\ \{P1,P2\}©(in_1 \circ in_2)$ Für $p \neq \langle\rangle\ :$ (2) $h(l,p)(\{P1toQ \mapsto in_1, P2toQ \mapsto in_2, DtoQ \mapsto \langle Next\rangle\} \circ s)$ $=\ \{QtoD \mapsto \langle ft.p, !Dto(ft.p)\rangle\} \circ h(j, (rt.p) \circ q)(s)$ (3) $h(l,\langle\rangle)(\{P1toQ \mapsto \langle\rangle, P2toQ \mapsto \langle\rangle, DtoQ \mapsto \langle Next\rangle\} \circ s)$ $=\ \{QtoD \mapsto \langle Empty\rangle\} \circ h(j,\langle\rangle)(s)$ (4a) $h(l,\langle\rangle)(\{PltoQ \mapsto \langle Pl, !DtoPl\rangle, PjtoQ \mapsto in_j, DtoQ \mapsto \langle Next\rangle\} \circ s)$ $=\ \{QtoD \mapsto \langle Pl, !DtoPl\rangle\} \circ h(j, \{P1,P2\}©in_j)(s)$ (4b) $h(l,\langle\rangle)(\{PltoQ \mapsto \langle\rangle, PjtoQ \mapsto \langle Pj, !DtoPj\rangle, DtoQ \mapsto \langle Next\rangle\} \circ s)$ $=\ \{QtoD \mapsto \langle Pj, !DtoPj\rangle\} \circ h(j,\langle\rangle)(s)$ (5) $h(l,p)(\{P1toQ \mapsto in_1, P2toQ \mapsto in_2, DtoQ \mapsto \langle\rangle\} \circ s) =\ \{QtoD \mapsto \langle\rangle\} \circ h(j, p \circ q)(s)$
wobei: $q = \{P1,P2\}©(in_l \circ in_j)$ und $j = (l \texttt{ mod } 2) + 1$

4.4.5 Zusammenspiel der Komponenten

Zu Beginn von Abschnitt 4.4 und in den Abschnitten 4.4.1 bis 4.4.4 haben wir das erweiterte verteilte System zur Modellierung der Prozessorverwaltung für zwei Prozesse entwickelt. Wie in Abschnitt 4.3.2 veranschaulichen wir nun das Zusammenspiel der Komponenten dieses erweiterten Systems anhand einiger EETs. Wir gehen entsprechend zu den in Abschnitt 4.3.2 gezeigten EETs davon aus, daß Prozeß $P1$ zur Berechnung von $\langle Step_1, Step_2, Step_3\rangle$ definiert ist. Für $P2$ sei die Berechnung von $\langle Step_1, Step_2\rangle$ festgelegt. Aufgrund der Tatsache, daß wir zur Darstellung aller möglichen Systemabläufe eine große Anzahl von Alternativen betrachten müssen, werden wir uns hier auf die Darstellung eines Beispielablaufs beschränken. Generell sind für vollständige Systemabläufe im betrachteten Beispielsystem folgende Punkte jeweils alternativ zu berücksichtigen:

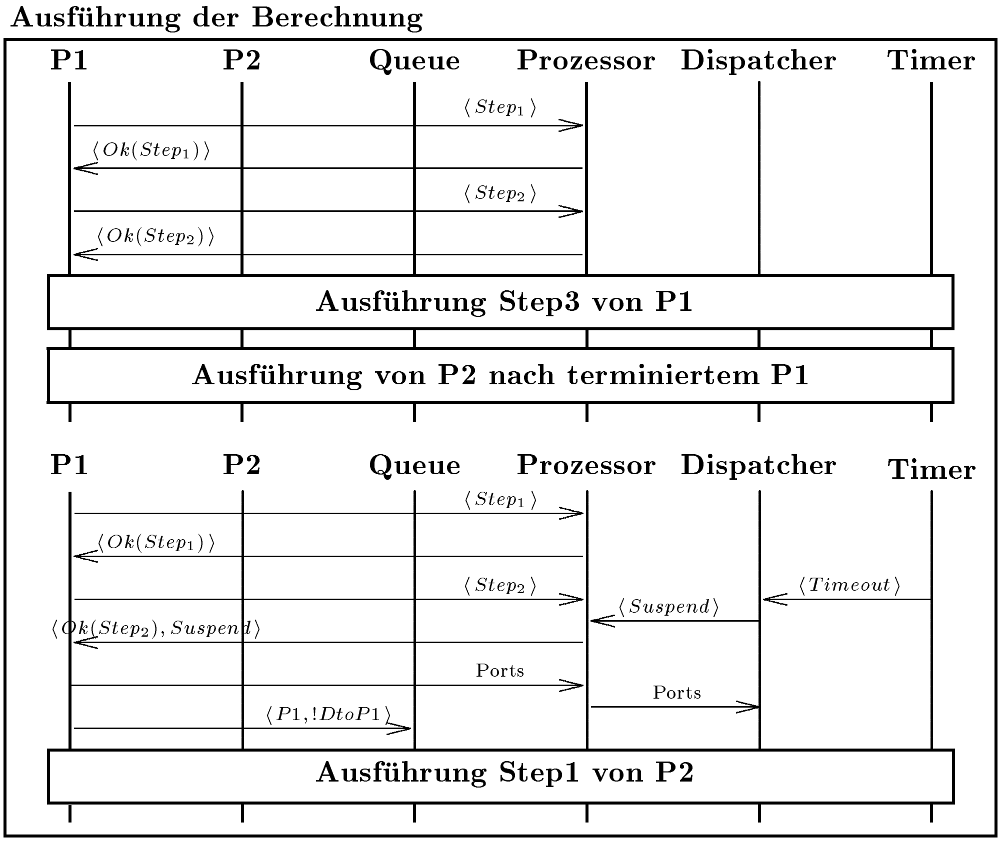
- $P1$ und $P2$ werden simultan gestartet und bewerben sich beide um den Prozessor, *oder* einer der beiden Prozesse wird zuerst gestartet, bewirbt sich um den Prozessor und beginnt mit der Durchführung der Berechnung.
- Für die Durchführung der Berechnungen ist zu berücksichtigen, daß jeder Prozeß nach jedem $Step_i$ suspendiert werden kann. Anschließend wird die Berechnung des nächsten Prozesses weitergeführt.

Wir zeigen ein EET für den Fall, daß $P1$ und $P2$ simultan gestartet werden und sich um den Prozessor bewerben. Aufgrund der Initialisierung des Systems wird der Prozessor zunächst $P1$ zugeteilt. Es ergibt sich das hier gezeigte EET, das sich leicht aus dem EET von Abbildung 4.3.3 herleiten läßt. Eine vertikale Linie, die $P2$ repräsentiert, ist hinzuzunehmen und entsprechend zum Start von $P1$ ist der Start von $P2$ darzustellen. Um die möglichen Alternativen zusätzlich einzuschränken, gehen wir weiter davon aus, daß das Quantum rrq der Zeitscheibe ausreicht, um mindestens die Schritte $Step_1$ und $Step_2$ von $P1$ vollständig auszuführen. Dies ist in der Box *Ausführung der Berechnung* dargestellt, die sich teilweise aus der Box *Ausführung der Berechnung* aus Abschnitt 4.3.2 herleiten läßt.

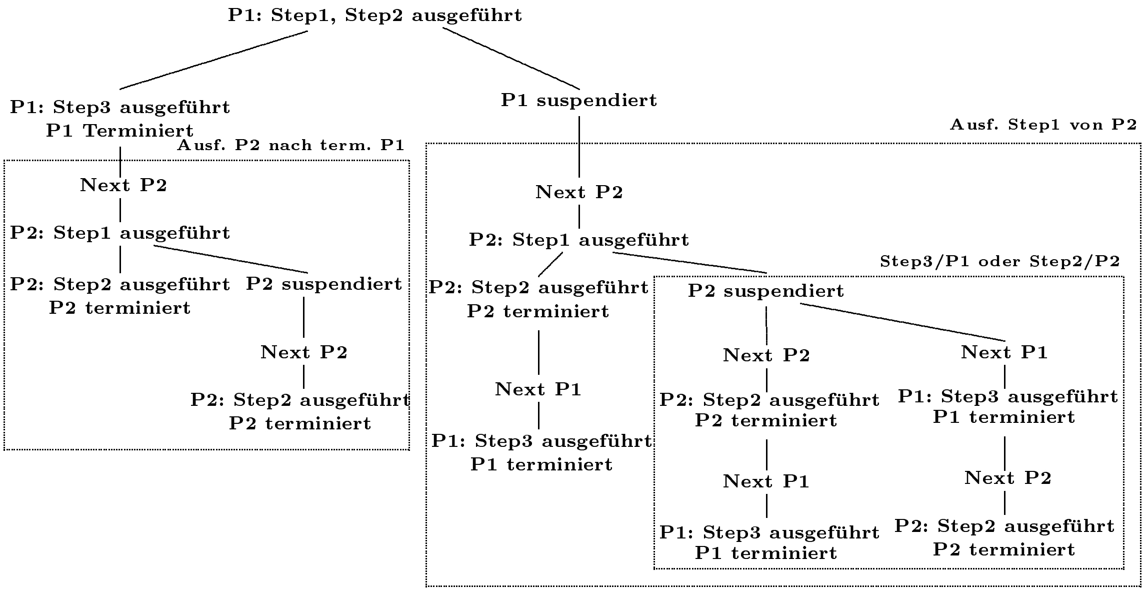


Die Alternativen, die im weiteren Systemablauf auftreten können, lauten:

1. Das Quantum rrq ist auch nach Ausführung von $Step_2$ noch nicht verstrichen; $P1$ kann seine Berechnung fortsetzen. Damit terminiert $P1$, und im System ist nur noch $P2$ vollständig auszuführen. Dieser mögliche Systemablauf wird durch das erste alternative EET in *Ausführung der Berechnung* dargestellt. Die dabei verwendete Box *Ausführung Step3 von P1* wurde in Abschnitt 4.3.2 auf Seite 53 gezeigt. Zur dieser Darstellung wird $P2$ hinzugenommen. Die Ausführung der Berechnung von $P2$ wird unter der Randbedingung fortgesetzt, daß $P1$ bereits terminiert ist.
2. $P1$ wird mit der Ausführung von $Step_2$ suspendiert, so daß $Step_3$ noch auszuführen ist. Dies zeigt das zweite alternative EET in *Ausführung der Berechnung*. Gemäß der Initialisierung unseres Systems wird ein Systemablauf so fortgesetzt, daß $P2$ an den Prozessor gebunden wird und die Ausführung der Berechnung von $P2$ beginnt.



Für die Abarbeitung der Berechnungen von $P1$ und $P2$ durch den Prozessor ergeben sich die in der folgenden Baumstruktur dargestellten Alternativen, die durch Bezeichner charakterisiert sind. Die Kästen markieren Boxen, die innerhalb der EETs auftreten, und die Kanten veranschaulichen die möglichen Alternativen innerhalb der Systemabläufe.



Diese Darstellung macht deutlich, welche tiefe Schachtelung von EETs sich bereits für diesen Beispielablauf ergibt. Die einzelnen Boxen können aus den bisher gezeigten EETs hergeleitet werden. Der Nachrichtenfluß in dieser graphischen Darstellung ist aufgrund der zahlreichen verwendeten Boxen nur noch schwer nachvollziehbar. Im folgenden werden wir auf die Veranschaulichung durch EETs verzichten.

4.5 Prozessorverwaltung für n Prozesse

In den Abschnitten 4.3 und 4.4 wurden die ersten Schritte zur formalen Modellierung der Prozessorverwaltung bewältigt. Für den Systemaufbau wurde eine vereinfachte Struktur gewählt, an der das für die Prozessorverwaltung wesentliche Verhalten demonstriert werden konnte. In Abschnitt 4.4 haben wir uns der zu behandelnden Konkurrenzsituation bzgl. des Prozessors auf einfache Weise genähert: *Zwei* Prozesse konkurrieren um den Prozessor. Die Queue sorgt dafür, daß kein Prozeß bevorzugt behandelt wird und die Zuteilung/Freigabe des Prozessors *fair* erfolgt. Beim Übergang zur Modellierung des auf zwei Prozesse erweiterten Systems konnten große Teile der Spezifikation aus Abschnitt 4.3 wiederverwendet werden. Mit diesen Vorbereitungen sind wir nun in der Lage, die allgemeine Aufgabenstellung der Prozessorverwaltung zu behandeln:

> Für ein Einprozessorsystem werden die Zuteilung und Freigabe des exklusiv nutzbaren Betriebsmittels „Prozessor" für endlich viele Prozesse modelliert.

Betrachten wir die einzelnen Komponenten ausgehend von den Abschnitten 4.3.1 und 4.4.1:

Prozessor: Im System ist *ein Prozessor* enthalten, der zu jedem Zeitpunkt genau einem Prozeß zugeteilt ist. Der Prozessor registriert nicht, an *welchen* Prozeß er aktuell gebunden ist, er führt die Instruktionen aus, die an ihn gesendet werden. Suspendierungs- und Terminierungsmeldungen werden weitergeleitet. Die Nachrichten, die der Prozessor empfängt, sind unabhängig von der Anzahl der Prozesse. An der Modellierung von PZ werden keine Änderungen vorgenommen.

Prozesse: Im System sind nun $n \in \mathbb{N}$ *Prozesse* Pi mit $i \in \{1, \ldots, n\}$ enthalten. Für jeden Pi ist eine Berechnung definiert, und Pi benötigt den Prozessor, um diese auszuführen. Da wir voneinander unabhängige Prozesse betrachten, kann jeder der n Prozesse entsprechend der Spezifikation von $P1$ modelliert werden. Alle Bezeichner sind so zu wählen, daß ihre Eindeutigkeit sichergestellt ist. Die Spezifikationen der Prozesse $P1$ und $P2$ aus den Abschnitten 4.3.3 und 4.4.2 werden vollständig wiederverwendet.

Dispatcher: Der *Dispatcher* stellt die Verbindung zwischen dem aktuell ausgewählten Prozeß und dem Prozessor her. Zur Erzeugung dieser Verbindung muß er mit dem ausgewählten Prozeß Pi verbunden sein. Den entsprechenden Port erhält er von der Queue. Es sind geringfügige Änderungen an der Spezifikation von D vorzunehmen.

Queue: In den Abschnitten 4.3.7 und 4.4.4 wurde gezeigt, daß die *Queue* auch Aufgaben eines Schedulers übernimmt. Da sich nun n Prozesse gleicher Priorität um den Prozessor bewerben, wird die Queue so spezifiziert, daß keine Anmeldung verloren geht und kein Prozeß bevorzugt wird. Eine derartige Modellierung wurde in Abschnitt 4.4.4 für das System mit zwei Prozessen erstellt. Dies muß auf n Prozesse verallgemeinert werden. Für die Erstellung der Spezifikation der Queue sind entsprechende Anpassungen der Spezifikation aus Abschnitt 4.4.4 nötig.

Die Spezifikation des *Timers*, an der bereits in Abschnitt 4.4 keine Änderung vorgenommen wurde, kann auch hier vollständig übernommen werden. Diese Erläuterungen zeigen, daß wir zur vollständigen Spezifikation des *Dispatching für n Prozesse* entsprechend zur Vorgehensweise in Abschnitt 4.4 nur gezielte Anpassungen an den Spezifikationen des Dispatchers und der Queue vornehmen. Die Anpassungen der Spezifikationen relativ zu den Abschnitten 4.4.3 und 4.4.4 und die Spezifikation des vollständigen *verteilten* Systems führen wir in den folgenden Abschnitten durch.

4.5.1 Ein System zur Prozessorverwaltung mit n Prozessen

Betrachten wir den hier erforderlichen Systemaufbau im Vergleich zu dem System, das wir in Abschnitt 4.4 modelliert haben. Das System besteht aus jeweils einem *Prozessor*, *Dispatcher*, *Timer* und einer *Queue*. Zur Vervollständigung des Systems für die allgemeine Aufgabenstellung sind nun dem System endlich viele Prozesse $P1$ bis Pn, mit $n \in I\!N$, hinzuzufügen. Alle Prozesse konkurrieren um den Prozessor.

Die Prozesse bilden die Schnittstelle des gesamten Systems zur Umgebung. Der Dispatcher ist mit der Queue, dem Timer und dem Prozessor verbunden. Entsprechend zu unseren bisher gezeigten Modellierungen ist der Dispatcher zeitweise mit dem ausgewählten Prozeß gekoppelt. Jeder Prozeß ist mit der Queue verbunden. Das System wird als mobiles Netz modelliert, in dem es bezogen auf die Zuteilung des Prozessors $n+1$ Phasen gibt: In *Phase 0* ist der Prozessor keinem der Prozesse zugeteilt. In *Phase i* mit $i \in \{1, \ldots, n\}$ ist Prozeß Pi an den Prozessor gebunden. Die Kopplung von Prozeß und Prozessor modellieren wir entsprechend zu dem in Abschnitt 4.4.1 gezeigten System durch das Erzeugen und Löschen der Lese- bzw. Schreibverbindungen mit den Kanälen $PtoPZ$ und $PZtoP$. Mit diesen Erläuterungen erhalten wir die in Abbildung 4.5.1 gezeigte Folge von SSDs, mit denen die oben genannten Phasen graphisch veranschaulicht werden.

Die Umsetzung des SSD zur initialen Vernetzung des Systems (Phase 0) liefert die in Abbildung 4.5.2 gezeigte ANDL-Spezifikation zum Netz `MultiP`. Die für `MultiP` gültige Menge aller Kanalbezeichner lautet N_{MultiP}. Es ist zu beachten, daß insbesondere die privaten Ports $?!DtoPi$ aller Prozesse Pi zu dieser Menge gehören.

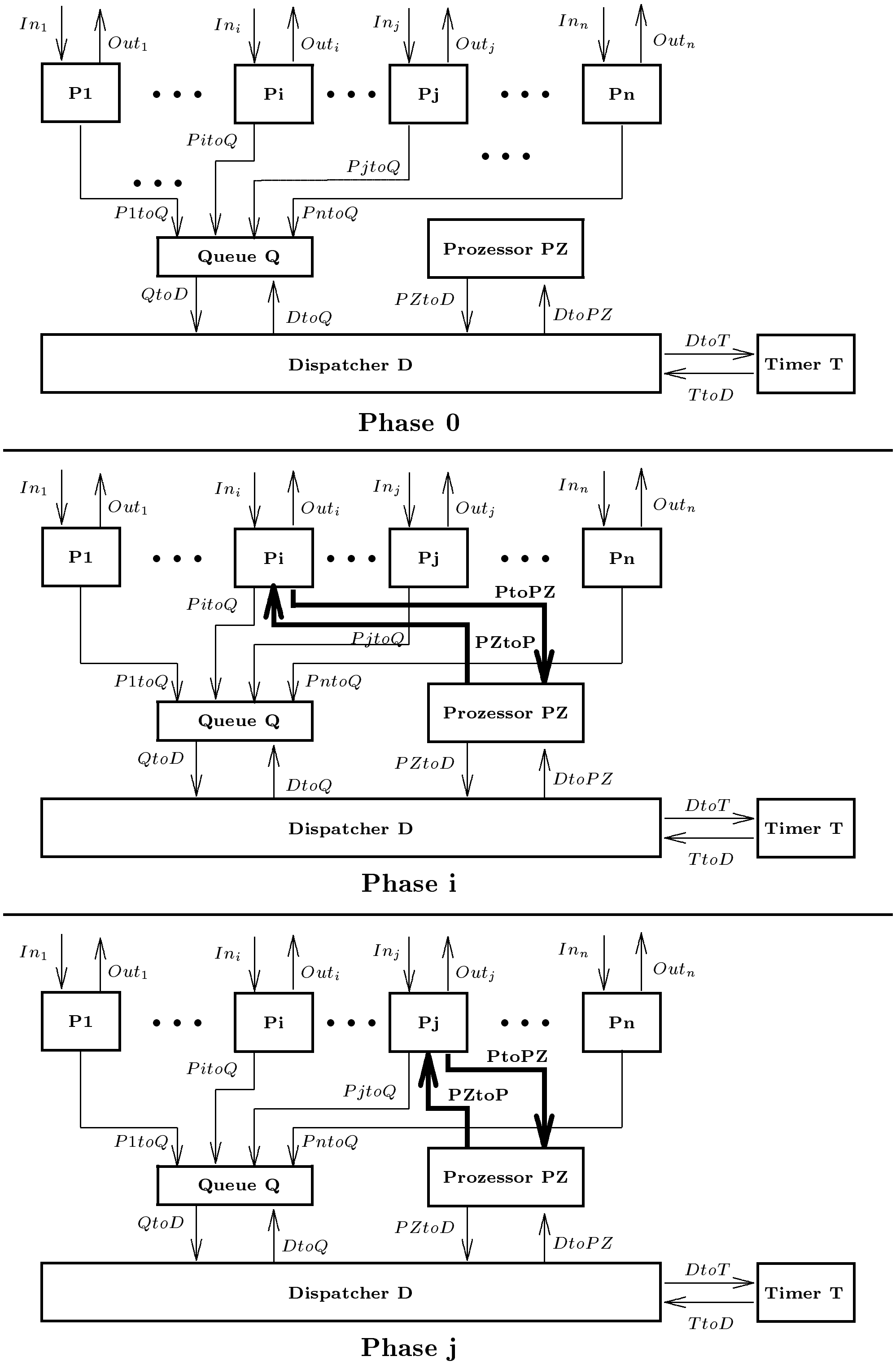


Abbildung 4.5.1: Prozessorzuteilung im System `MultiP` mit n Prozessen

agent `MultiP`

input channels $In_1 : S_{In_1}, \ldots, \boldsymbol{In_i} : \boldsymbol{S_{In_i}}, \ldots, \boldsymbol{In_n} : \boldsymbol{S_{In_n}}$

output channels $Out_1 : S_{Out_1}, \ldots, \boldsymbol{Out_i} : \boldsymbol{S_{Out_i}}, \ldots, \boldsymbol{Out_n} : \boldsymbol{S_{Out_n}}$

is network

$$
\begin{array}{lcl}
\ll Out_1, P1toQ \gg & = & \mathtt{P1} \ll In_1 \gg \; ; \\
\vdots & \vdots & \vdots \\
\ll \boldsymbol{Out_i, PitoQ} \gg & = & \boldsymbol{Pi} \ll \boldsymbol{In_i} \gg \; ; \\
\vdots & \vdots & \vdots \\
\ll \boldsymbol{Out_n, PntoQ} \gg & = & \boldsymbol{Pn} \ll \boldsymbol{In_n} \gg \; ; \\
\ll QtoD \gg & = & \mathtt{Q} \ll P1toQ, \ldots, \boldsymbol{PitoQ}, \ldots, \boldsymbol{PntoQ}, DtoQ \gg \; ; \\
\ll DtoQ, DtoT, DtoPZ \gg & = & \mathtt{D} \ll QtoD, TtoD, PZtoD \gg \; ; \\
\ll PZtoD \gg & = & \mathtt{PZ} \ll DtoPZ \gg \; ; \\
\ll TtoD \gg & = & \mathtt{T} \ll DtoT \gg \; ;
\end{array}
$$

end `MultiP`

Abbildung 4.5.2: ANDL-Spezifikation für das System `MultiP` mit n Prozessen

Im folgenden betrachten wir den verallgemeinerten Dispatcher und die erweiterte Queue. Auf die Spezifikation der Prozesse verzichten wir, da diese Spezifikationen entsprechend zum Prozeß P2 in Abschnitt 4.4.2 erstellt werden können.

4.5.2 Der Dispatcher für n Prozesse

Die Spezifikation des Dispatchers wird nur geringfügig geändert. Im Vergleich zum Dispatcher des Systems mit zwei Prozessen, siehe Abschnitt 4.4.3, kann D jeweils kurzfristig eine Kanalverbindung $DtoPi$ zum ausgewählten Prozeß Pi schreibend nutzen, um den Verbindungsaufbau zwischen Pi und PZ zu initiieren. D wird als mobile Komponente modelliert. Die den Kanälen $DtoQ$, $PZtoD$, $DtoPZ$, $TtoD$, $DtoT$, $PtoPZ$ sowie $PZtoP$ zugeordneten Nachrichtentypen ändern sich außer der Anpassung an $?!N_{MultiP}$ nicht und sind in Tabelle 4.3.3 von Seite 62 aufgeführt. Über den Kanal $QtoD$ können alle Identifikatoren Pi empfangen werden. Diese Erweiterung ist in Tabelle 4.5.1 auf Seite 83 enthalten. Ausgehend vom SSD für das System `MultiP`, siehe Abbildung 4.5.1, erhalten wir die in Abbildung 4.5.3 gezeigte ANDL-Spezifikation für den Dispatcher zur Verwaltung von n Prozessen.

Bei der Spezifikation des Verhaltens von D werden alle Fälle betrachtet, in denen der Dispatcher eine Nachricht Pi und einen Port $!DtoPi$ mit $i \in \{1, \boldsymbol{\ldots, n}\}$ von der Queue erhält. Dieses veränderte Verhalten des Dispatchers wird durch Verallgemeinerung der Spezifikation von (1) und (6) auf Seite 72 angegeben.

(1) D wird gestartet, wenn er über Kanal $QtoD$ eine erste Sequenz $\langle Pi, !DtoPi \rangle$, mit $i \in \{1, \boldsymbol{\ldots, n}\}$, empfängt. D initiiert den Verbindungsaufbau mit den Kanälen $PtoPZ$ und $PZtoP$ zwischen Pi und dem Prozessor, sendet die Nachricht rrq über Kanal $DtoT$ und geht in den Zustand *bound* über.

(6) Erhält D im Zustand *free* eine Sequenz $\langle Pi, !DtoPi \rangle$, mit $i \in \{1, \boldsymbol{\ldots}, \boldsymbol{n}\}$, so initiiert D den Verbindungsaufbau zwischen Pi und dem Prozessor mit den Kanälen $PtoPZ$ und $PZtoP$ über die Kanäle $DtoPi$ bzw. $DtoPZ$, sendet die Nachricht rrq über Kanal $DtoT$ und geht in den Zustand *bound* über.

Von der formalen Spezifikation des Dispatchers auf Seite 72 können die Gleichungen (1) und (6) übernommen werden. Der Index i wird von „2" zu „n" verallgemeinert. Die Formalisierungen ergeben sich durch Umsetzung der textuellen Beschreibung nach Schema (2.13) von Seite 26 und entsprechend zu (1) bzw. (6) von Seite 63. Zur Formalisierung verweisen wir auf Seite 72, in der „Für $i \in \{1, 2\}$" durch „Für $i \in \{1, \boldsymbol{\ldots}, \boldsymbol{n}\}$" ersetzt werden muß.

agent D
  input channels $QtoD : S_{QtoD},\ TtoD : S_{TtoD},\ PZtoD : S_{PZtoD}$
  output channels $DtoQ : S_{DtoQ},\ DtoT : S_{DtoT}, DtoPZ : S_{DtoPZ}$
  private channels $PtoPZ : S_{PtoPZ},\ PZtoP : S_{PZtoP}$
is basic
  f_{D} $\boxed{\text{mit der Spezifikation von Seite 63 und 72 verallgemeinert auf } i \in \{1, \ldots, n\}}$
end D

Abbildung 4.5.3: ANDL-Spezifikation von D bei n Prozessen

4.5.3 Der Warteraum für n Prozesse

Die Spezifikation von Q muß erweitert werden, da sich n Prozesse gleicher Priorität um den Prozessor bewerben. Abbildung 4.5.1 zeigt, daß alle Pj für $j \in \{1, \ldots, n\}$ mit Q verbunden sind. Zur Festlegung der Nachrichtentypen gelten die in Tabelle 4.5.1 gezeigten Erweiterungen der Spezifikation von Q in den Abschnitten 4.3.7 und 4.4.4.

Kanal n	Nachrichtenmengen S_n	
$P1toQ$	$\{P1\}$	$\cup\ ?!N_{\boldsymbol{MultiP}}$
$\vdots$	$\vdots$	
$\boldsymbol{PntoQ}$	$\boldsymbol{\{Pn\}}$	
$DtoQ$	$\{Next\}$	
$QtoD$	$\{Empty\} \cup \{P1, \boldsymbol{\ldots}, \boldsymbol{Pi}, \boldsymbol{\ldots}, \boldsymbol{Pn}\}$	

Tabelle 4.5.1: Nachrichtentypen für Q mit n Prozessen

Die Spezifikation der Queue wird so erweitert, daß jede Anforderung des Prozessors registriert und kein Prozeß Pj mit $j \in \{1, \ldots, n\}$ bevorzugt wird. Dies erreichen wir durch

Erweiterung der Modellierung der Queue von Abschnitt 4.4.4, indem jeweils *alle* an den Kanälen $PjtoQ$ mit $j \in \{1, \ldots, n\}$ empfangenen Anforderungen registriert werden. Zusätzlich werden sie, jeweils zyklisch verschoben, gemäß der FIFO-Strategie abgespeichert.

agent Q
  input channels $P1toQ : S_{P1toQ}, \boldsymbol{\ldots, PitoQ : S_{PitoQ}}$,
  $\boldsymbol{\ldots, PntoQ : S_{PntoQ}}, DtoQ : S_{DtoQ}$
  output channels $QtoD : S_{QtoD}$
  private channels $\emptyset$
is basic
  f_{Q} $\boxed{\text{mit der Spezifikation von Seite 85}}$
end Q

Abbildung 4.5.4: ANDL-Spezifikation von Q für n Prozesse

Die *richtige* Reihenfolge wird durch Verallgemeinerung der in Abschnitt 4.4.4 vorgestellten Vorgehensweise modelliert: Der Reihenfolgeparameter bleibt erhalten. Zunächst wird die Anforderung von $P1$ registriert, falls sie vorliegt. Mit jedem Zeitintervall wird der Parameter um 1 inkrementiert, wobei wir „mod $\boldsymbol{n}$“ statt „mod 2“ verwenden. Der Parameter nimmt zyklisch die Werte „1“ bis „n“ an, alle Eingabekanäle $PjtoQ$ werden behandelt, und jeder Kanal ist nach dem Verstreichen von n Zeitintervallen wieder *an der Reihe*.

Wir zeigen die Spezifikation der Anforderungen (1) und (4); vgl. Seite 74. Alle weiteren Spezifikationen ergeben sich mit den hier gezeigten Gleichungen und gemäß Abschnitt 4.4.4. Mit $p \in \{P1, \ldots, Pn\}^*$ bezeichnen wir den internen Speicher von Q.

(1) Q wird gestartet, wenn mindestens ein Prozeß den Prozessor anfordert. Die Anforderung, die gemäß der numerischen Reihenfolge als erste auftritt, wird über Kanal $QtoD$ gesendet. Alle übrigen Anforderungen werden gespeichert. Zusätzlich wird registriert, daß im nächsten Zeitintervall Kanal $P2toQ$ bevorzugt wird.

(4) Es gilt $p = \langle\rangle$. Q erhält die Anfrage *Next* über Kanal $DtoQ$. Auf dem priorisierten Kanal $PltoQ$ wird keine Anforderung des Prozessors empfangen. Fordert mindestens ein weiterer Prozeß den Prozessor an, wird die gemäß der aktuell gültigen *und* der numerischen Reihenfolge nächste Anforderung über Kanal $QtoD$ gesendet.

Die Punkte (2), (3), (4a) und (5) der Seiten 74 und 76 können bis auf die Anpassung an die n Kanäle $PjtoQ$ und die korrekte Abspeicherung der Pj wiederverwendet werden.

Erweiterung der Spezifikation für f_Q von Seite 67

$$\forall s \in \prod_{n \in N_{MultiP}} [S_n^*],\ i,k,l \in \{1,\boldsymbol{\ldots},\boldsymbol{n}\},\ in_k \in \{\langle Pk, !DtoPk\rangle, \langle\rangle\},\ p,q,r \in \{P1,\ldots,Pn\}^* \ :$$

$$\exists h \in (\{1,\boldsymbol{\ldots},\boldsymbol{n}\} \times \{P1,\boldsymbol{\ldots},\boldsymbol{Pn}\}^*) \ \rightarrow\ Type_Q \ :$$

Für $\boldsymbol{\bigvee_{i=1}^{n} (in_i = \langle Pi, !DtoPi\rangle)}$ **:**

(1) $f_Q(\{P1toQ \mapsto in_1, \boldsymbol{\ldots}, \boldsymbol{PntoQ \mapsto in_n}\} \circ s) \ = \ \{QtoD \mapsto \langle ft.r, !Dto(ft.r)\rangle\} \circ h(2, rt.r)(s)$

wobei $r \ = \ \{P1,\ldots,\boldsymbol{Pn}\} \copyright (in_1 \circ \boldsymbol{\ldots} \circ \boldsymbol{in_n})$

Für $in_l = \langle\rangle$ und $\exists i \in \{1,\ldots,n\} \setminus \{l\} \ : \ in_i = \langle Pi, !DtoPi\rangle \ :$

(4b) $h(l,\langle\rangle)(\{P1toQ \mapsto in_1, \boldsymbol{\ldots}, \boldsymbol{PntoQ \mapsto in_n}, DtoQ \mapsto \langle Next\rangle\} \circ s)$
$\qquad = \ \{QtoD \mapsto \langle ft.q, !Dto(ft.q)\rangle\} \circ h(j, rt.q)(s)$

wobei $j \ = \ (l \ \mathtt{mod}\ \boldsymbol{n}) + 1$ und

$\boldsymbol{q \ = \ \{P1,\ldots,Pn\} \copyright (in_l \circ in_{(l \ \mathtt{mod}\ n)+1} \circ in_{(l+1 \ \mathtt{mod}\ n)+1} \circ \ldots \circ in_{((l+(n-2)) \ \mathtt{mod}\ n)+1})}$

Funktionsweise der Queue

Das Prinzip, nach dem die Identifikatoren zyklisch abgespeichert werden, erläutern wir anhand eines Beispiels. Q sei eine Komponente mit vier Eingabekanäle $ToQ1$ bis $ToQ4$, wobei die auf diesem Weg empfangenen Nachrichten nach dem oben beschriebenen Prinzip abgespeichert werden. Über Kanal $ToQ1$ wird das Zeichen „A“ und über die Kanäle $ToQ2$, $ToQ3$ und $ToQ4$ jeweils „B“, „C“ bzw. „D“ gesendet.

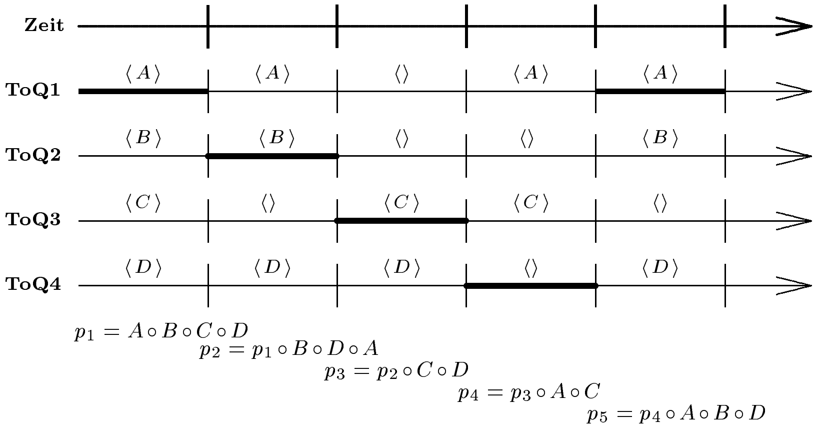


Abbildung 4.5.5: Zyklisches Abspeichern für die Queue

Abbildung 4.5.5 veranschaulicht das Abspeichern der Nachrichten für fünf Zeitintervalle. Die oberste Achse stellt das Voranschreiten der Zeit dar, und die weiteren vier Achsen zeigen die über den einen Kanal eintreffenden Ströme. Mit p_i für $i \in \{1, \ldots, 5\}$ wird der aktuelle Wert des internen Speichers der Queue angezeigt. Die **markierten** Zeitabschnitte zeigen den Kanal, der im aktuellen Zeitintervall zuerst behandelt wird.

Mit dem Start eines Systemablaufs wird die über Kanal $ToQ1$ eintreffende Nachricht zuerst und alle weiteren Nachrichten gemäß der numerischen Reihenfolge abgespeichert, zunächst die Nachricht von Kanal $ToQ2$ gefolgt von der Nachricht auf Kanal $ToD3$ und abschließend die Nachricht von Kanal $ToQ4$. Es ergibt sich der gezeigte Wert von p_1. Der interne Reihenfolgeparameter wird auf „2" gesetzt. Im zweiten Zeitintervall wird die über Kanal $ToQ2$ eintreffende Nachricht zuerst und die von Kanal $ToQ1$ abschließend abgespeichert. Nach diesem Prinzip wird im dritten Zeitintervall Kanal $ToQ3$ zuerst betrachtet. Das weitere Vorgehen kann anhand der graphischen Darstellung nachvollzogen werden.

4.6 Zentrales Dispatching von m Prozessoren

Bisher haben wir Systeme modelliert, in denen für die Ausführung der Prozesse *genau ein* Prozessor zur Verfügung steht. In diesem Abschnitt und in Abschnitt 4.7 spezifizieren wir Systeme, in denen mehrere Prozessoren verwaltet werden. Da die prinzipielle Vorgehensweise bei der Spezifikationsentwicklung anhand der bisher gezeigten Modellierungen bereits bekannt ist, werden wir direkt auf ein System mit $m < n$ Prozessoren eingehen. Im zunächst entwickelten System gehen wir davon aus, daß die Prozessoren von einem zentralen Dispatcher verwaltet werden. Damit lautet die allgemeine Aufgabenstellung:

> Für ein Multiprozessorsystem mit einem zentralen Dispatcher werden die Zuteilung und Freigabe des Betriebsmittels „Prozessor" für n Prozesse modelliert.

Ausgehend von dieser Aufgabenstellung und den vorher modellierten Einprozessorsystemen betrachten wir den in Abschnitt 4.5.1 entwickelten Systemaufbau und die dort spezifizierten Komponenten. Gemäß der von uns allgemein gewählten Vorgehensweise nehmen wir nur Veränderungen an Spezifikationen von solchen Komponenten vor, deren Verhalten direkt von der neuen Aufgabenstellung betroffen ist.

Prozessoren: Im System sind nun m *Prozessoren* enthalten, die jeweils von einem Prozeß genutzt werden können. Jeder Prozessor führt die Berechnung des Prozesses aus, dem er aktuell zugeteilt ist, und leitet Suspendierungs- und Terminierungsmeldungen weiter. Die Prozessoren sind unabhängig voneinander. Die Modellierung aller Prozessoren wird aus Abschnitt 4.3.4 übernommen, wobei sicherzustellen ist, daß alle Kanalbezeichner eindeutig gewählt sind.

Prozesse: Im System sind n *Prozesse* enthalten, deren Spezifikation aus Abschnitt 4.3.3 erweitert wird. Jeder Prozeß kann mit jedem Prozessor gekoppelt werden.

Timer: Für jeden Prozessor ist jeweils ein *Timer* zuständig, der den Ablauf der Zeitscheibe mißt. Für das System sind m Timer zu modellieren. Sie sind mit dem Dispatcher verbunden und werden jeweils mit rrq gestartet. Die Spezifikation des Timers aus Abschnitt 4.3.6 wird mit den Anpassungen an die Bezeichner wiederverwendet.

Queue: Die *Queue* ist dafür zuständig, die Prozessoranforderungen der Prozesse entgegenzunehmen. Sie ist direkt mit dem Dispatcher verbunden und reagiert nur auf Nachrichten des Dispatchers bzw. der Prozesse. Die in Abschnitt 4.5.3 erstellte Modellierung der Queue kann vollständig wiederverwendet werden.

Dispatcher: Der *Dispatcher* ist für die Verwaltung der Prozessoren zuständig und koppelt Prozesse mit freien Prozessoren. Er ist

- zeitweise mit jedem anfordernden Prozeß Pi durch eine Kanalverbindung, die er schreibend nutzen darf, und
- mit allen Prozessoren PZj mit $j \in \{1, \ldots, m\}$ jeweils durch eine Kanalverbindung, die er schreibend, und eine, die er lesend nutzt,

verbunden. Die Zuteilung von Prozessoren und Prozessen erfolgt durch die Erzeugung entsprechender Kanalverbindungen. Die Spezifikation des Dispatchers wird an die $m > 1$ Prozessoren angepaßt.

Im folgenden zeigen wir, wie die Spezifikationen der Prozesse und des Dispatchers aus den Abschnitten 4.3.3 und 4.5.3 erweitert werden.

4.6.1 Ein System für zentrales Dispatching

Zu dem Systemaufbau aus Abschnitt 4.5.1 nehmen wir nun $m - 1$ weitere *Prozessoren* $PZ2$ bis PZm und die entsprechenden *Timer* $T2$ bis Tm hinzu. In den bisher gezeigten Spezifikationen benennen wir PZ in $PZ1$ und T in $T1$ um. n Prozesse konkurrieren um die m Prozessoren, die vom Dispatcher verwaltet werden. Für jeden Prozessor ist ein Zeitgeber zuständig. Alle Timer sind mit dem Dispatcher verbunden.

Zur Steigerung der Übersichtlichkeit verzichten wir in Abbildung 4.6.1 darauf, alle n Prozesse, m Prozessoren und m Timer darzustellen. Die Prozesse werden durch einen Kasten „Prozesse“, die Prozessoren und Timer jeweils durch den Kasten „Prozessoren“ bzw. „Timer“ dargestellt, in denen jeweils exemplarisch ein Prozeß Pi, ein Prozessor PZj und ein Timer Tj gezeigt wird. Die Prozesse bilden mit ihren Ein- und Ausgabekanälen In_i bzw. Out_i die Schnittstelle zur Umgebung. Entsprechend zu den bisher gezeigten Modellierungen koppelt der Dispatcher bei Bedarf einen ausgewählten Prozeß mit dem zugeordneten Prozessor. Hierfür erhält er zeitweise das Schreibrecht an einer Kanalverbindung *DtoPi* zum ausgewählten Prozeß Pi. Jeder Prozeß kann im Verlauf der Durchführung seiner Berechnung mit jedem der Prozessoren verbunden sein. Abbildung 4.6.1 zeigt beispielhaft die Phase, in der Prozeß Pi an Prozessor PZj gebunden ist. Initial ist kein Prozeß an einen Prozessor gebunden.

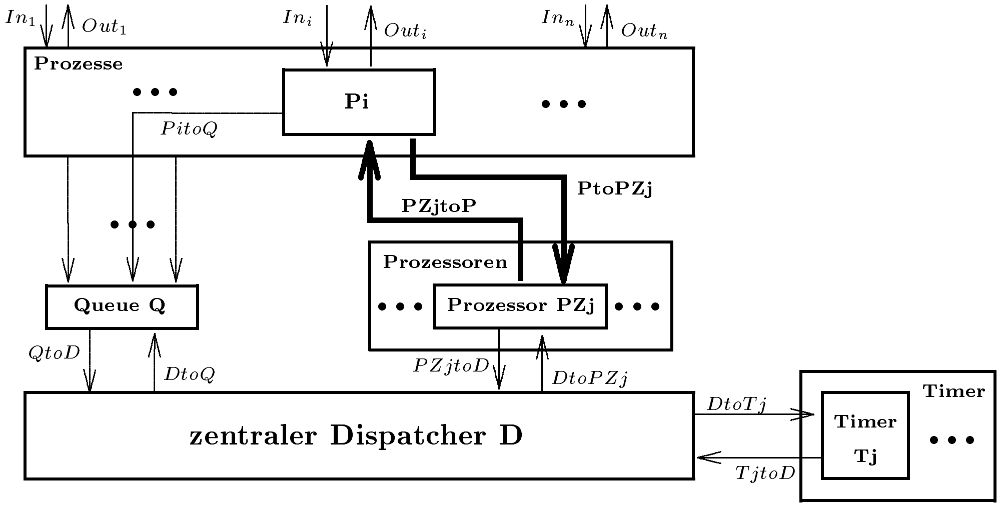


Abbildung 4.6.1: Zentrales Dispatching für m Prozessoren

agent `MultiZPZ`
  input channels $In_1 : S_{In_1}, \ldots, In_n : S_{In_n}$
  output channels $Out_1 : S_{Out_1}, \ldots, Out_n : S_{Out_n}$
is network

$$
\begin{array}{lcl}
\ll Out_1, P1toQ \gg & = & \mathtt{P1} \ll In_1 \gg \;; \\
\vdots & \vdots & \vdots \\
\ll Out_n, PntoQ \gg & = & \mathtt{Pn} \ll In_n \gg \;; \\
\ll QtoD \gg & = & \mathtt{Q} \ll P1toQ, \ldots, PntoQ, DtoQ \gg \;; \\
\ll DtoQ, DtoT1, \boldsymbol{\ldots}, \boldsymbol{DtoTm}, & & \\
\quad DtoPZ1, \boldsymbol{\ldots}, \boldsymbol{DtoPZm} \gg & = & \mathtt{D} \ll QtoD, T1toD, \boldsymbol{\ldots}, \boldsymbol{TmtoD}, \\
 & & \quad PZ1toD, \boldsymbol{\ldots}, \boldsymbol{PZmtoD} \gg \;; \\
\ll PZ1toD \gg & = & \mathtt{PZ1} \ll DtoPZ1 \gg \;; \\
\vdots & \vdots & \vdots \\
\boldsymbol{\ll PZmtoD \gg} & \boldsymbol{=} & \boldsymbol{PZm \ll DtoPZm \gg \;;} \\
\ll T1toD \gg & = & \mathtt{T1} \ll DtoT1 \gg \;; \\
\vdots & \vdots & \vdots \\
\boldsymbol{\ll TmtoD \gg} & \boldsymbol{=} & \boldsymbol{Tm \ll DtoTm \gg \;;}
\end{array}
$$

end `MultiZPZ`

Abbildung 4.6.2: ANDL-Spezifikation für das *zentrale* Multiprozessorsystem `MultiZPZ`

Wir erhalten die in Abbildung 4.6.2 gezeigte ANDL-Spezifikation für das System `MultiZPZ`. Die Menge der Kanalbezeichner bezeichnen wir mit $N_{MultiZPZ}$, die sich aus der Menge N_{MultiP} durch Hinzunahme der Kanalbezeichner der Verbindungen zu den *neuen* Prozessoren und Timern ergibt.

Im folgenden werden wir die Spezifikationen des für das System `MultiZPZ` erweiterten Dispatchers und der Prozesse, die an jeden der Prozessoren gekoppelt werden können, entwickeln. Auf die Spezifikation der Timer verzichten wir, da diese entsprechend zum Timer aus Abschnitt 4.3.6 erstellt werden können.

4.6.2 Prozesse im System mit m Prozessoren

Bei der Anpassung der Spezifikation eines Prozesses Pi an m zentral verwaltete Prozessoren ist zu berücksichtigen, daß jeder Prozeß an jeden Prozessor gebunden werden kann. Die Spezifikation von Abschnitt 4.6.3 wird zeigen, daß der Dispatcher über die entsprechenden privaten Ports verfügt. In der Spezifikation eines Prozesses, vergleiche Abschnitt 4.3.3, sind Gleichungen enthalten, die die Fälle spezifizieren, in denen Prozeß und Prozessor gebunden sind. Dabei wird auf die Kanalbezeichner $PtoPZ$ und $PZtoP$ Bezug genommen. Im System `MultiZPZ` können die Kanalbezeichner $PtoPZj$ und $PZjtoP$ mit $j \in \{1, \ldots, m\}$ auftreten. Die Spezifikationen werden dementsprechend angepaßt.

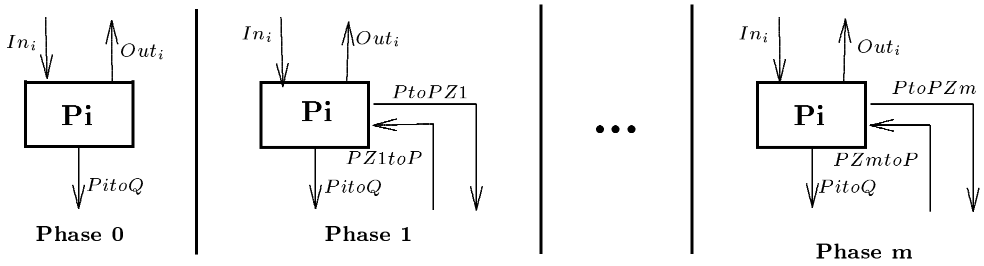


Abbildung 4.6.3: SSDs der möglichen $m + 1$ Prozeßphasen

Abbildung 4.6.3 zeigt die möglichen Phasen, die ein Prozeß Pi bezogen auf die Kopplung an einen Prozessor durchlaufen kann. In *Phase 0* ist er an keinen Prozessor gebunden; dies entspricht der Initialisierungsphase und allen Phasen, in denen Pi auf die Zuteilung eines Prozessors wartet. In *Phase 1* ist Pi der Prozessor $PZ1$ und in *Phase m* der Prozessor PZm zugeteilt. Die Nachrichtentypen für die Kanäle in_i, Out_i und $PitoQ$ entnehmen wir der Tabelle 4.3.1, wobei sie insbesondere an die erweiterte Menge $N_{MultiPZ}$ angepaßt werden müssen. Entsprechend kann die in Abbildung 4.3.5 gezeigte ANDL-Spezifikation, angepaßt an Pi, übernommen werden. Für die Festlegung von f_{Pi} sind die Spezifikationen auf Seite 57 und die Anpassungen auf Seite 90 zu finden.

Die Ausgangsbasis bildet das auf Seite 56 informell beschriebene und auf Seite 57 spezifizierte Verhalten eines Prozesses: Bei der Erweiterung muß berücksichtigt werden, daß ein

Prozeß vom Dispatcher die Zugriffsrechte zu Kanalverbindungen zu *jedem* Prozessor PZj mit $j \in \{1, \ldots, m\}$ erhalten kann. Dies muß bei der Beschreibung und Spezifikation des Verhaltens entsprechend erfaßt werden.

(2) Erhält Pi im Zustand *ready* die Sequenz $\langle \boldsymbol{?PZjtoP}, \boldsymbol{!PtoPZj}, \boldsymbol{!DtoPi} \rangle$ über Kanal $DtoPi$, so sendet Pi über Kanal $\boldsymbol{PtoPZj}$ den nächsten Berechnungsschritt und geht in den Zustand *busy* über.

(3) Erhält Pi im Zustand *busy* über Kanal $PtoPZj$ ein $\langle Ok(Step) \rangle$, verbleibt er im Zustand *busy* und sendet den nächsten Berechnungsschritt über Kanal $PtoPZj$.

(4) Erhält Pi im Zustand *busy* zusätzlich zum $\langle Ok(Step) \rangle$ die Nachricht $Suspend$, werden die Verbindungen zum **zugeordneten** Prozessor $\boldsymbol{PZj}$ gelöscht. Pi sendet $\langle Pi, !DtoPi \rangle$ über Kanal $PitoQ$ und geht in den Zustand *ready* über.

(5) Erhält Pi das „Ok“ zum letzten Berechnungsschritt, so sendet Pi die Nachricht $Term$ über Kanal $\boldsymbol{PtoPZj}$, löscht die Verbindungen zu PZj, sendet die Nachricht $OutputPi$ über Kanal Out_i und geht in den Zustand $term$ über.

Diese textuelle Beschreibung wird wie folgt umgesetzt:

Erweiterung der Spezifikation für f_{Pi} von Seite 57

Für $\boldsymbol{j \in \{1, \ldots, m\}}$:

$$(2)\quad h(ready,p)(\{DtoPi \mapsto \langle \boldsymbol{?PZjtoP}, \boldsymbol{!PtoPZj}, \boldsymbol{!DtoPi} \rangle\} \circ s) = \{\boldsymbol{PtoPZj} \mapsto \langle ft.p \rangle\} \circ h(busy,p)(s)$$

Für $\#p > 1$:

$$(3)\quad h(busy,p)(\{\boldsymbol{PZjtoP} \mapsto \langle Ok(ft.p) \rangle\} \circ s) = \{\boldsymbol{PtoPZj} \mapsto \langle ft.rt.p \rangle\} \circ h(busy,rt.p)(s)$$

$$(4)\quad h(busy,p)(\{\boldsymbol{PZjtoP} \mapsto \langle Ok(ft.p), Suspend \rangle\} \circ s) = \{\boldsymbol{PtoPZj} \mapsto \langle \boldsymbol{?PZjtoP}, \boldsymbol{!PtoPZj} \rangle, PitoQ \mapsto \langle Pi, !DtoPi \rangle\} \circ h(ready,rt.p)(s)$$

Für $\#p = 1$:

$$(5a)\quad h(busy,p)(\{\boldsymbol{PZjtoP} \mapsto \langle Ok(ft.p) \rangle\} \circ s) = \{\boldsymbol{PtoPZj} \mapsto \langle Term, \boldsymbol{?PZjtoP}, \boldsymbol{!PtoPZj} \rangle, Out_i \mapsto \langle OutputPi \rangle\} h(term, \langle\rangle)(s)$$

$$(5b)\quad h(busy,p)(\{\boldsymbol{PZjtoP} \mapsto \langle Ok(ft.p), Suspend \rangle\} \circ s) = \{\boldsymbol{PtoPZ} \mapsto \langle Term, \boldsymbol{?PZjtoP}, \boldsymbol{!PtoPZj} \rangle, Out_i \mapsto \langle OutputPi \rangle\} \circ h(term, \langle\rangle)(s)$$

4.6.3 Der zentrale Dispatcher für m Prozessoren

Zu Beginn von Abschnitt 4.6 wurde erklärt, wie die Spezifikation des Dispatchers aus Abschnitt 4.5.2 zu erweitern ist. Zusätzlich zur erweiterten Schnittstelle, die durch alle neuen Komponenten PZj und Tj bestimmt wird, wird das Verhalten so erweitert, daß der Dispatcher für die Zuteilung jedes Prozessors an anfordernde Prozesse und die Einhaltung der Zeitscheibe in Verbindung mit den Timern sorgt. Der Dispatcher durchläuft mehrere Phasen, die dadurch charakterisiert sind, daß entweder kein Prozessor, nur ein Teil der Prozessoren oder alle Prozessoren aktuell mit Prozessen gekoppelt sind. Der Dispatcher verfügt initial über die Menge privater Ports, die für den Aufbau der Verbindung jedes Prozessors zu einem Prozeß nötig ist. *Phase 0* sei die Initialisierungsphase und bestimmt auch die Schnittstelle für den Fall, daß kein Prozessor an einen Prozeß gebunden ist.

Für die Festlegung der Nachrichtentypen sind die Kanäle für die Verbindungen zu den weiteren Prozessoren, den Timern und für die Ergänzungen der privaten Ports und $?!PZjtoP$ und $?!PtoPZj$ zu berücksichtigen. Die Kanalbezeichner des bisher modellierten Dispatchers werden um einen *Zähler j* mit $j \in \{1, \ldots, m\}$ erweitert. Für jeden Kanal sind die Nachrichtentypen gültig, die dem entsprechenden Kanal des Dispatchers für einen Prozessor zugeordnet sind. Aus diesem Grund erhalten wir die für den hier modellierten Dispatcher gültigen Nachrichtenmengen aus den Tabellen 4.3.3 und 4.4.1 auf den Seiten 62 bzw. 72. Es ist zu beachten, daß über Kanal *QtoD* alle Prozeßidentifikatoren $P1$ bis Pn empfangen werden können und allen Kanälen die erweiterte Menge $?!N_{MultiZPZ}$ zuzuordnen ist.

Der Dispatcher ist in der Lage, m Prozessoren in einem System mit n Prozessen gleicher Priorität so zu verwalten, daß ein freier Prozessor einem Prozeß zugeteilt wird, wenn es anfordernde Prozesse gibt. Hierbei gibt es für die Prozesse und Prozessoren keine feste Zuteilung. Jedem Prozeß kann jeder Prozessor zugeteilt werden, falls dieser frei ist. D erhält in jedem Zeitintervall von zahlreichen Komponenten Information, auf die er reagieren muß, und die nicht verloren gehen dürfen. Für den Dispatcher gilt:

1. D erfragt bei der Queue den nächsten Prozeßidentifikator durch *aktives Warten*, koppelt den ausgewählten Prozeß mit einem freien Prozessor mittels der Ports und startet den zugehörigen Timer. Zusätzlich müssen Suspendierungs- und Reset-Signale an Timer bzw. Prozessoren weitergegeben werden.

2. D reagiert auf Timeout-Signale aller Timer und sendet an die entsprechenden Prozessoren ein Suspendierungssignal.

3. D reagiert auf Terminierungs-Signale und auf den Empfang der Ports und registriert die freigewordenen Prozessoren.

Gemäß den oben beschriebenen Aufgaben werden wir den zentralen Dispatcher zur Verwaltung von m Prozessoren nicht als Basiskomponente, sondern als verteiltes System modellieren. Der Dispatcher besteht aus Unterkomponenten, die für die oben genannten drei Teilbereiche zuständig sind. Diese Vorgehensweise werden wir generell beibehalten: Läßt sich die Funktionalität einer Komponente in mehrere Aufgaben aufteilen, die separat behandelt werden können und in Kooperation die geforderte Funktionalität gewährleisten, modellieren wir eine Komponente als Netz. Auf diese Weise bleiben die Formalisierungen *handlicher*. Der zentrale Dispatcher setzt sich aus folgenden Komponenten zusammen: *DT* für die Entgegennahme der *Timeout* von allen Timern, *DP* für die Schnittstelle zu

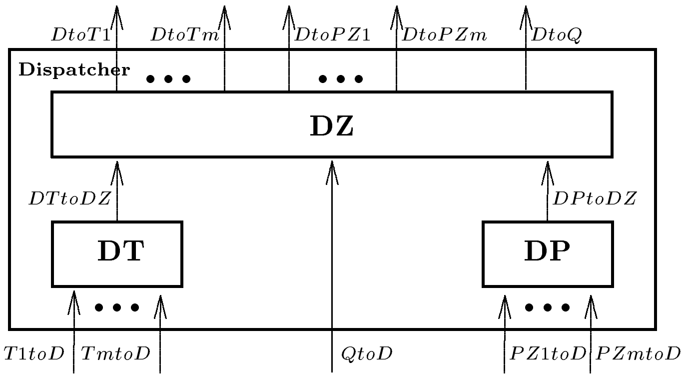


Abbildung 4.6.4: Glass-Box-Sicht auf den zentralen Dispatcher

den Prozessoren und den hier eintreffenden Suspendierungs- und Terminierungsmeldungen und die zentrale Schaltstelle DZ für die Kopplung von Prozessoren und ausgewählten Prozessen. Für den Dispatcher gilt die in Abbildung 4.6.5 gezeigte ANDL-Spezifikation. Die Subkomponenten werden in den folgenden Abschnitten spezifiziert.

agent D
  input channels $QtoD : S_{QtoD}, T1toD : S_{T1toD}, \boldsymbol{\ldots, TmtoD : S_{TmtoD}}$,
  $PZ1toD : S_{PZ1toD}, \boldsymbol{\ldots, PZmtoD : S_{PZmtoD}}$
  output channels $DtoT1 : S_{DtoT1}, \boldsymbol{\ldots, DtoTm : S_{DtoTm}}$,
  $DtoPZ1 : S_{DtoPZ1}, \boldsymbol{\ldots, DtoPZm : S_{DtoPZm}}$
is network

$$\begin{aligned}
\ll DtoQ, DtoT1, \ldots, DtoTm,& \\
DtoPZ1, \ldots, DtoPZm \gg \;&=\; \mathtt{DZ} \ll QtoD, DTtoDZ, DPtoDZ \gg; \\
\ll DTtoDZ \gg \;&=\; \mathtt{DT} \ll T1toD, \ldots, TmtoD \gg; \\
\ll DPtoDZ \gg \;&=\; \mathtt{DP} \ll PZ1toD, \ldots, PZmtoD \gg;
\end{aligned}$$

end D

Abbildung 4.6.5: ANDL-Spezifikation des zentralen Dispatchers

4.6.3.1 Die zentrale Schaltstelle DZ des Dispatchers

Die zentrale Schaltstelle des Dispatchers ist die Komponente DZ. Sie wird als mobile Komponente modelliert, da sie für die Initiierung des Verbindungsaufbaus zwischen Prozessen und Prozessoren zuständig ist. Ausgehend von dem in Abbildung 4.6.4 gezeigten SSD erhalten wir die ANDL-Spezifikation von DZ in Abbildung 4.6.6.

agent DZ
input channels $QtoD : S_{QtoD}, DTtoDZ : S_{DTtoDZ}, DPtoDZ : S_{DPtoDZ}$
output channels $DtoQ : S_{DtoQ}, DtoT1 : S_{DtoT1}, \ldots, DtoTm : S_{DtoTm},$
$DtoPZ1 : S_{DtoPZ1}, \ldots, DtoPZm : S_{DtoPZm}$
private channels $PtoPZ1 : S_{PtoPZ1},\ PZ1toP : S_{PZ1toP}, \ldots,$
$PtoPZm : S_{PtoPZm},\ PZmtoP : S_{PZmtoP}$
is basic
f_{DZ} $\boxed{\text{mit der Spezifikation von Seite 95}}$
end DZ

Abbildung 4.6.6: ANDL-Spezifikation von DZ

Für die bekannten Verbindungskanäle zu den Timern, Prozessoren, Prozessen und der Queue können wir die Nachrichtentypen aus den entsprechenden Modellierungen entnehmen. Für die beiden neu eingeführten internen Kanalverbindungen von DT und DP zu DZ müssen die Nachrichtentypen festgelegt werden. Sie sind in Tabelle 4.6.1 aufgeführt.

Kanal n	Nachrichtenmengen S_n
$DTtoDZ$	$\{\langle M \rangle \mid M \in \mathcal{P}(\{1,\ldots,m\})\} \cup\ ?!N_{MultiPZ}$
$DPtoDZ$	$\{\langle M \rangle \mid M \in \mathcal{P}(\{1,\ldots,m\})\} \cup\ ?!N_{MultiPZ}$

Tabelle 4.6.1: Nachrichtentypen für DZ

Zur textuellen Beschreibung des Verhaltens von DZ verallgemeinern wir die Modellierung des Dispatchers von System MultiP. Dabei ist zu berücksichtigen, daß die beiden Komponenten DT und DP zusätzlich eingeführt wurden, um die entsprechend benötigte Information zu sammeln. Im folgenden steht $p \in \{1, \ldots, m\}^*$ für die Liste der freien Prozessoren.

(1) DZ wird durch eine erste Sequenz $\langle Pi, !DtoPi \rangle$ über Kanal $QtoD$ gestartet, initiiert die Kopplung zwischen Pi und dem Prozessor $PZ1$ mittels der Kanäle $PtoPZ1$ und $PZ1toP$, sendet die Nachricht rrq über Kanal $DtoT1$, über Kanal $DtoQ$ die Nachricht $Next$ und geht in den Zustand $p = 2 \circ \ldots \circ m$ über.

(2) Es sind mehrere Prozessoren frei, d.h. es gilt: $\#p > 1$. Erhält DZ über Kanal $QtoD$ eine Sequenz $\langle Pi, !DtoPi \rangle$, so initiiert DZ den Verbindungsaufbau zwischen Pi und dem Prozessor PZk mittels der Kanäle $PtoPZk$ und $PZktoP$ über die Kanäle $DtoPi$ bzw. $DtoPZk$ und sendet die Nachricht rrq über Kanal $DtoTk$ und die Nachricht $Next$ über Kanal $DtoQ$. Der Prozessor PZk ergibt sich aus der Liste der freien Prozessoren p durch $k = ft.p$.

Die aktuell frei werdenden Prozessoren und abgelaufenen Zeitscheiben müssen erfaßt werden. Die Information erhält DZ über die Kanäle $DTtoDZ$ und $DPtoDZ$.

- An alle Prozessoren PZj_i, die aufgrund einer abgelaufenen Zeitscheibe suspendiert werden müssen, und die nicht aufgrund der Terminierung des Prozesses frei geworden sind, wird die Nachricht *Suspend* geschickt.
- An alle Timer Tl_i, deren zugeordneter Prozessor PZl_i eine Terminierungsmeldung geschickt hat und deren Zeitscheibe nicht abgelaufen ist, wird ein $\langle\, Reset \,\rangle$ geschickt.
- Alle Prozessoren, die im aktuellen Zeitintervall frei geworden sind, entweder durch Terminierung des Prozesses oder durch eine erfolgreiche Suspendierung, werden in die Liste der freien Prozessoren von DZ aufgenommen.

(3) Es ist *ein* Prozessor frei, d.h. es gilt $\#p = 1$. Erhält DZ über Kanal *QtoD* die Sequenz $\langle\, Pi, !DtoPi \,\rangle$, so initiiert DZ die Kopplung zwischen Pi und dem Prozessor PZk mit $k = ft.p$.

(a) Erhält DZ zusätzlich über Kanal *DPtoDZ* die Nachricht $\emptyset$, so geht er in den Zustand „$\langle\rangle$“ über. Aufgrund der aktuell abgelaufenen Zeitscheiben werden *Suspend*-Nachrichten verschickt.

(b) Erhält DZ über Kanal *DPtoDZ* eine Sequenz $\langle\, M_P, PortList \,\rangle$, sendet DZ die Nachricht $Next$ über Kanal *DtoQ* und geht in den Zustand $FreePZ$ über. Zusätzlich werden *Suspend*- bzw. *Reset*-Nachrichten an Prozessoren bzw. Timer gesendet. (Vergleiche auch Punkt (2).)

(4) Erhält DZ im Zustand $p \neq \langle\rangle$ über Kanal *QtoD* die Nachricht *Empty*, wird die Nachricht $Next$ über Kanal *DtoQ* gesendet. Zusätzlich werden *Suspend*- bzw. *Reset*-Nachrichten an Prozessoren bzw. Timer gesendet und die frei gewordenen Prozessoren in p aufgenommen. (Vergleiche auch Punkt (2).)

(5) Erhält DZ im Zustand $p = \langle\rangle$ über Kanal *DPtoDZ* die Sequenz $\langle\, M_P, PortList \,\rangle$, sendet er die Nachricht $Next$ über Kanal *DtoQ* und geht in den Zustand $FreePZ$ über. Zusätzlich werden *Suspend*- bzw. *Reset*-Nachrichten an Prozessoren bzw. Timer gesendet. (Vergleiche auch Punkt (2).)

Mit (1) wird berücksichtigt, daß wir ein Multiprozessorsystem mit $m > 1$ Prozessoren modellieren. Da mit diesem ersten Schritt nur ein Prozessor belegt ist, kann der nächste Prozeß direkt aus der Queue entnommen werden. Schritt (2) beschreibt das Verhalten für den Fall, daß mehrere Prozessoren frei sind und eine Anforderung nach einem Prozessor empfangen wird. Dem Prozeß wird ein Prozessor zugeteilt. (3) beschreibt die Situation, in der noch genau ein Prozessor frei ist und eine Anforderung empfangen wird. Wir unterscheiden (3a), wobei im aktuellen Zeitintervall kein weiterer Prozessor frei geworden ist. Hier kann keine nächste Anforderung aus der Queue entnommen werden. In (3b) sind weitere Prozessoren frei geworden, ein nächster Prozeß wird bei der Queue angefordert und die Liste der freien Prozessoren entsprechend aktualisiert. Mit (4) und (5) modellieren wir das aus den vorhergehenden Spezifikationen bekannte *aktive Warten*, wobei wir mit (5) den

Fall explizit betrachten, in dem ein nächster Prozeß erst dann wieder angefordert wird, wenn Prozessoren frei werden.

Funktionsgleichungen für f_{DZ}

$$\forall s \in \prod_{n \in N_{MultiZPZ}} [S_n^*],\ rrq, k \in I\!N,\ p, FreePZ \in \{1,\ldots,m\}^*,\ i \in \{1,\ldots,n\}$$
$$M_T, M_P \in \mathcal{P}(\{1,\ldots,m\}),\ PortList \in \{?!PZjtoP, ?!PtoPZj\}^* \ :$$
$$\exists h \in (\{1,\ldots,m\}^*) \to Type_{DZ} \ :$$

(1) $f_{DZ}(\{QtoD \mapsto \langle Pi, !DtoPi\rangle\} \circ s)$
$= \{DZtoT1 \mapsto \langle rrq\rangle, DtoPi \mapsto \langle ?PZ1toP, !PtoPZ1, !DtoPi\rangle,$
$DtoPZ1 \mapsto \langle !PZ1toP, ?PtoPZ1\rangle, DtoQ \mapsto \langle Next\rangle\} \circ h(\langle 2,\ldots,m\rangle)(s)$

Für $\#p > 1$ und $k = ft.p$:

(2) $h(p)(\{QtoD \mapsto \langle Pi, !DtoPi\rangle, DTtoDZ \mapsto \langle M_T\rangle, DPtoDZ \mapsto \langle M_P, PortList\rangle\} \circ s)$
$= \{DtoTk \mapsto \langle rrq\rangle, DtoQ \mapsto \langle Next\rangle,\ \mathtt{PiAlloc},\ \mathtt{Suspend},\ \mathtt{Reset}\} \circ h(rt.p \circ FreePZ)(s)$

Für $\#p = 1$ und $k = ft.p$:

(3a) $h(p)(\{QtoD \mapsto \langle Pi, !DtoPi\rangle, DTtoDZ \mapsto \langle M_T\rangle, DPtoDZ \mapsto \langle \emptyset\rangle\} \circ s)$
$= \{DtoTk \mapsto \langle rrq\rangle,\ \mathtt{PiAlloc},\ \mathtt{Suspend}\} \circ h(\langle\rangle)(s)$

Für $\#p = 1,\ \#PortList \neq 0$ und $k = ft.p$:

(3b) $h(p)(\{QtoD \mapsto \langle Pi, !DtoPi\rangle, DTtoDZ \mapsto \langle M_T\rangle, DPtoDZ \mapsto \langle M_P, PortList\rangle\} \circ s)$
$= \{DtoTk \mapsto \langle rrq\rangle, DtoQ \mapsto \langle Next\rangle,\ \mathtt{PiAlloc},\ \mathtt{Suspend},\ \mathtt{Reset}\} \circ h(FreePZ)(s)$

Für $\#p \neq 0$ und $PortList \neq \langle\rangle$:

(4) $h(p)(\{QtoD \mapsto \langle Empty\rangle, DTtoDZ \mapsto \langle M_T\rangle, DPtoDZ \mapsto \langle M_P, PortList\rangle\} \circ s)$
$= \{DtoQ \mapsto \langle Next\rangle, \mathtt{Suspend}, \mathtt{Reset}\} \circ h(p \circ FreePZ)(s)$

(5) $h(\langle\rangle)(\{QtoD \mapsto \langle\rangle, DTtoDZ \mapsto \langle M_T\rangle, DPtoDZ \mapsto \langle M_P, PortList\rangle\} \circ s)$
$= \{DtoQ \mapsto \langle Next\rangle, \mathtt{Suspend}, \mathtt{Reset}\} \circ h(FreePZ)(s)$

wobei : $\mathtt{PiAlloc} = DtoPi \mapsto \langle ?PZktoP, !PtoPZk, !DtoPi\rangle, DtoPZk \mapsto \langle !PZktoP, ?PtoPZk\rangle$
$\mathtt{Suspend} = DtoPZj_1 \mapsto \langle Suspend\rangle, \ldots, DtoPZj_m \mapsto \langle Suspend\rangle$
$\mathtt{Reset} = DtoTl_1 \mapsto \langle Reset\rangle, \ldots, DtoTl_m \mapsto \langle Reset\rangle$
$\{j_1,\ldots,j_m\} = M_T \setminus M_P,$ und $\{l_1,\ldots,l_m\} = M_P \setminus M_T,$
$\forall l \in \{1,\ldots,m\} \ : \ (\#l © FreePZ = 1 \ \Leftrightarrow \ \#\{!PZltoP\} © PortList = 1) \ \wedge$
$(\#l © FreePZ = 0 \ \Leftrightarrow \ \#\{!PZltoP\} © PortList = 0)$

Die *Nummern* der Prozessoren, die suspendiert werden müssen, ergeben sich aus den Menge, die DZ von den Komponenten DT und DP erhält. Die Mengen M_T bzw. M_P charakterisieren die Timer, die ein *Timeout* melden, bzw. die Prozessoren, deren zugeordnete Prozesse terminiert sind. An die Prozessoren, die nicht durch die Menge M_P identifiziert werden, wird eine Suspendierungsnachricht gesendet. Falls ein Prozessor eine Suspendierung in Verbindung mit der Terminierung des ihm zugeordneten Prozesses abschließt, wird an den zugehörigen Timer ein *Reset* geschickt; dies ist durch die Spezifikation eines Timers abgedeckt, siehe Abschnitt 4.3.6, Seite 65, Punkt (4b). Die *Nummern* der Prozessoren, deren Bindung zu einem Prozeß erfolgreich gelöst wurde, ergeben sich aus den Nachrichten, die die Komponente DP an DZ sendet. Ein Prozessor ist dann frei geworden, falls er Ports an den Dispatcher zurücksendet; mit dieser Information können die zugehörigen Nummern in die Liste der freien Prozessoren aufgenommen werden.

4.6.3.2 Die Komponente DT

Die Komponente DT übernimmt die Registrierung aller Timeoutsignale durch die m Timer Tj und leitet die Information an die Zentrale des Dispatchers DZ weiter. DZ benötigt die Information darüber, welche Timer ein Timeout gesendet haben. Da die Tj numeriert sind, reicht es aus, wenn DT eine Menge $M_T \subseteq \{1, \ldots, m\}$ an DZ sendet. Hierbei ist ein $j \in \{1, \ldots, m\}$ genau dann in M_T enthalten, wenn der Timer Tj im aktuellen Zeitintervall ein *Timeout* gesendet hat. Abbildung 4.6.7 zeigt die ANDL-Spezifikation für DT.

agent DT
  input channels $T1toD : S_{T1toD}, \ldots, TmtoD : S_{TmtoD}$
  output channels $DTtoDZ : S_{DTtoDZ}$
  private channels $\emptyset$
is basic
  f_{DT} $\boxed{\text{mit der Spezifikation von Seite 96}}$
end DT

Abbildung 4.6.7: ANDL-Spezifikation von DT

Für das Verhaltens wird gefordert:

(1) Erhält DT über mindestens einen Kanal $T1toD$ bis $TmtoD$ die Nachrichten *Timeout*, wird $\langle M_T \rangle$ über Kanal $DTtoDZ$ gesendet.

(2) Erhält DT über die Kanäle $T1toD$ bis $TmtoD$ die Sequenz $\langle\rangle$, wird $\langle \emptyset \rangle$ über Kanal $DTtoDZ$ gesendet.

Die Fälle (1) und (2) können wir zusammenfassend durch eine Gleichung spezifizieren.

Funktionsgleichungen für f_{DT}
$\forall s \in \prod_{n \in N_{MultiZPZ}} [S_n^*],\ k \in \{1, \ldots, m\},\ in_k \in \{\langle Timeout \rangle, \langle\rangle\},\ M_T \in \mathcal{P}(\{1, \ldots, m\}) \ :$
(1) + (2) $f_{DT}(\{T1toD \mapsto in_1, \ldots, TmtoD \mapsto in_m\} \circ s) \ = \ \{DTtoDZ \mapsto \langle M_T \rangle\} \circ f_{DT}(s)$
wobei $\forall j \in \{1, \ldots, m\} \ : \ j \in M_T \ \Leftrightarrow \ in_j = \langle Timeout \rangle$

4.6.3.3 Die Komponente DP

Die Komponente DP registriert alle Terminierungs- und Suspendierungsmeldungen der Prozessoren und leitet die entsprechenden Informationen an DZ weiter. DZ benötigt Informationen darüber, welche Prozesse aktuell terminiert sind, da in diesen Fällen möglicherweise ein *Reset* an den entsprechenden Timer gesendet werden muß. DP sendet über Kanal $DPtoDZ$ sowohl eine Menge, die die frei gewordenen Prozessoren identifiziert, als auch eine Liste von Ports, die die Verbindungen zwischen Prozessen und Prozessoren darstellen. Abbildung 4.6.8 zeigt die ANDL-Spezifikation für DP.

agent DP
  input channels $PZ1toD \;:\; S_{PZ1toD}, \ldots, PZmtoD \;:\; S_{PZmtoD}$
  output channels $DPtoDZ \;:\; S_{DPtoDZ}$
  private channels $\emptyset$
is basic
  f_{DP} $\boxed{\text{mit der Spezifikation von Seite 97}}$
end DP

Abbildung 4.6.8: ANDL-Spezifikation von DP

Das Verhalten wird beschrieben durch:

(1) Erhält DP über mindestens einen Kanal $PZ1toD$ bis $PZmtoD$ die Nachricht *Term* oder eine Liste von Ports, werden über Kanal $DPtoDZ$ eine Menge M_T sowie eine Liste von Ports gesendet.

(2) Erhält DP über alle Kanäle $PZ1toD$ bis $PZmtoD$ die Sequenz $\langle\rangle$, wird $\langle\emptyset\rangle$ über Kanal $DPtoDZ$ gesendet.

Funktionsgleichungen für f_{DP}

$$\forall s \in \prod_{n \in N_{MultiZPZ}} [S_n^*],\; k \in \{1,\ldots,m\},\; M_T \in \mathcal{P}(\{1,\ldots,m\}$$
$$in_k \in \{\langle Term, ?!PZktoP, ?!PtoPZk\rangle, \langle ?!PZktoP, ?!PtoPZk\rangle, \langle\rangle\} \;:$$

Für $\bigvee_{i=1}^{m} (in_i \neq \langle\rangle) \;:$

(1) $f_{DP}(\{PZ1toD \mapsto in_1, \ldots, PZmtoD \mapsto in_m\} \circ s) \;=\; \{DPtoDZ \mapsto \langle M_T, Port\rangle\} \circ f_{DP}(s)$

wobei $\forall j \in \{1,\ldots,m\} \;:\; j \in M_T \;\Leftrightarrow\; \#(\{Term\}©in_j) = 1$

und $Port = \{?!PZjtoP, ?!PtoPZj\}©(in_1 \circ \ldots \circ in_m)$

(2) $f_{DP}(\{PZ1toD \mapsto \langle\rangle, \ldots, PZmtoD \mapsto \langle\rangle\} \circ s) \;=\; \{DPtoDZ \mapsto \langle\emptyset\rangle\} \circ f_{DP}(s)$

4.7 Verteiltes Dispatching von m Prozessoren

Mit Abschnitt 4.6 wurde gezeigt, wie ein Dispatcher für ein Multiprozessorsystem spezifiziert wird, der die Prozessoren zentral verwaltet. Der zentrale Dispatcher muß sowohl auf die *Timeout*- als auch auf die *Terminate*-Nachrichten reagieren. Aufgrund der dabei auftretenden zahlreichen Varianten erhält der Dispatcher zusätzliche Funktionalität, die bei der Spezifikation mit FOCUS durch explizit aufgeführte Fallunterscheidungen sichtbar wird. Eine Alternative zur Verwaltung mehrerer Prozessoren besteht darin, jedem Prozessor einen eigenen Dispatcher zuzuordnen und somit eine verteilte Variante zu modellieren. Wir behandeln folgende Aufgabenstellung:

> Für ein Multiprozessorsystem werden die Zuteilung und Freigabe des Betriebsmittels „Prozessor" für n Prozesse durch verteiltes Dispatching modelliert.

Es wird ein System modelliert, in dem jedem Prozessor ein separater Dispatcher zugeteilt ist. Jeder Dispatcher verfügt über einen zugeordneten Timer, mit dem der Ablauf der Zeitscheibe gemessen wird. Wir erhalten ein System, in dem das Subsystem bestehend aus *Timer*, *Dispatcher* und *Prozessor* des in Abschnitt 4.5 gezeigten Systems vervielfacht wird. Im System sind $m \in I\!N$ Prozessoren enthalten. Alle Prozessoren und Dispatcher arbeiten unabhängig voneinander. Jeder Dispatcher ist mit der Queue verbunden, in der die Anforderungen rechenbereiter Prozesse eingetragen sind. Für die *Dispatcher*, *Timer* und *Prozessoren* wird das Verhalten aus den Abschnitten 4.5.2, 4.3.6 und 4.3.4 übernommen und die entsprechende Spezifikation wiederverwendet. Bei der Vervielfältigung der Komponenten muß die Eindeutigkeit der Kanalbezeichner gewährleistet sein. Entsprechend sind die Spezifikationen in den oben genannten Abschnitten anzupassen und für das hier modellierte System zu verwenden. Für die *Prozesse* und die *Queue* gilt:

Prozesse: Im System sind n Prozesse enthalten. Zur Ausführung der Berechnung, für die der Prozeß definiert ist, benötigt er *einen* der im System enthaltenen m Prozessoren. Ein Prozeß macht mit der Anforderung eines Prozessors einen privaten Port öffentlich, der vom Dispatcher für die Kopplung von Prozeß und Prozessor verwendet wird. Dieser Port ist unabhängig von der Anzahl der im System enthaltenen Dispatcher. Prozesse werden entsprechend der in Abschnitt 4.3.3 entwickelten Spezifikation und der in Abschnitt 4.6.2 gezeigten Erweiterung auf m Prozessoren modelliert.

Queue: Im System bewerben sich n Prozesse gleicher Priorität um einen der m Prozessoren. Die Queue hat folgende Aufgaben:

1. Kein Prozeßidentifikator darf verloren gehen und kein Prozeß bei der Zuteilung der Prozessoren bevorzugt werden.
2. Alle Prozessoren müssen ausgelastet sein, die Anfragen der m Dispatcher werden immer beantwortet, solange es anfordernde Prozesse gibt.

Die Queue aus Abschnitt 4.5.3, deren Spezifikation den oben genannten Punkt 1. bereits erfüllt, ist um das mit 2. beschriebene Verhalten zu erweitern.

4.7.1 Das System für verteiltes Dispatching

Zum Systemaufbau aus Abschnitt 4.6.1 nehmen wir nun $n-1$ Subsysteme $PZjDj$ mit $j \in \{1,\ldots,m\}$ hinzu. Jedes derartige Subsystem besteht aus einem Prozessor PZj , einem Dispatcher Dj und einem Timer Tj.

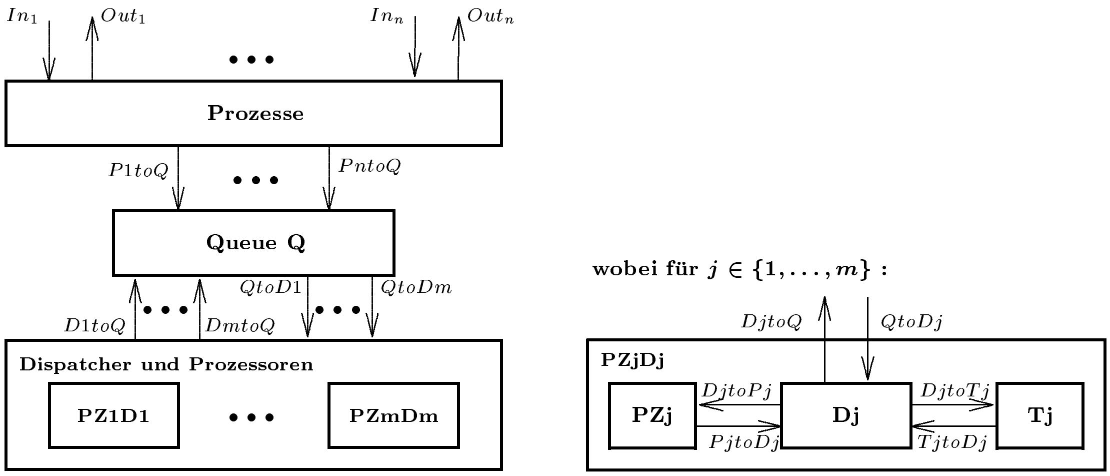


Abbildung 4.7.1: Verteiltes Dispatching für m Prozessoren

Das in Abbildung 4.7.1 dargestellte System durchläuft viele Phasen. Zur Steigerung der Übersichtlichkeit verzichten wir darauf, alle n Prozesse und m Subsysteme im SSD darzustellen und nutzen die Möglichkeit der hierarchischen Darstellung. Die im System auftretenden Prozesse sind im Kasten „Prozesse" und die oben genannten Subsysteme im Kasten „Dispatcher und Prozessoren" dargestellt. Alle Subsysteme arbeiten unabhängig voneinander und stimmen sich mittels der zentralen Queue ab. Jeder Dispatcher ist jeweils durch einen Kanal, den er lesend und einen Kanal, den er schreibend nutzen kann, mit der Queue verbunden. Entsprechend zu unseren bisherigen Modellierungen führt der Dispatcher die Zuteilung zwischen einem ausgewählten Prozeß und dem zugeordneten Prozessor mittels der Erzeugung von Kanalverbindungen durch. Da jeder Prozeß im Verlauf der Ausführung seiner Berechnung an jeden der Prozessoren gebunden sein kann, ergeben sich die Phasen, die das System durchläuft. Im SSD verzichten wir auf die explizite graphische Darstellung dieser Phasen. Die Prozesse bilden durch ihre Ein- und Ausgabekanäle In_i bzw. Out_i die Schnittstelle des Systems zur Umgebung, deren charakteristische Vernetzungsstruktur inzwischen bekannt ist. Die initiale Phase entspricht der Vernetzungsstruktur, in der kein Prozeß an einen Prozessor gebunden ist. Die Umsetzung dieses SSD ergibt die in Abbildung 4.7.2 gezeigte ANDL-Spezifikation für das System `MultiVPV`.

Die Menge der Kanalbezeichner $N_{MultiVPV}$ ergibt sich aus den bisher vorgestellten Modellierungen, aus der ANDL-Spezifikation in Abbildung 4.7.2 und durch eine Erweiterung der Menge N_{MultiP}. Entsprechend zu Abschnitt 4.6 sind im hier modellierten System für jeden

Prozessor die privaten Kanäle $PZjtoP$ und $PtoPZj$ mit $j \in \{1, \ldots, m\}$ zur Kopplung an einen Prozessor definiert.

$$
\begin{array}{l}
\textsf{agent } \texttt{MultiVPV} \\
\quad \textsf{input channels} \quad In_1 \ : \ S_{In_1}, \ldots, In_n \ : \ S_{In_n} \\
\quad \textsf{output channels} \quad Out_n \ : \ S_{Out_1}, \ldots, Out_n \ : \ S_{Out_n} \\
\textsf{is network} \\
\quad \begin{array}{lcl}
\ll Out_1, P1toQ \gg & = & \texttt{P1} \ll In_1 \gg \ ; \\
\vdots & \vdots & \vdots \\
\ll Out_n, PntoQ \gg & = & \texttt{Pn} \ll In_n \gg \ ; \\
\ll QtoD1, \boldsymbol{\ldots, QtoDm} \gg & = & \texttt{Q} \ll P1toQ, \ldots, PntoQ, \\
 & & \quad D1toQ, \boldsymbol{\ldots, DmtoQ} \gg \ ; \\
\ll D1toQ, D1toT1, D1toPZ1 \gg & = & \texttt{D1} \ll QtoD1, T1toD1, PZ1toD1 \gg \ ; \\
\vdots & \vdots & \vdots \\
\ll \boldsymbol{DmtoQ,} & & \\
\quad \boldsymbol{DmtoTm, DmtoPZm} \gg & \boldsymbol{=} & \boldsymbol{Dm} \ll \boldsymbol{QtoDm,} \\
 & & \quad \boldsymbol{TmtoDm, PZmtoDm} \gg \ ; \\
\ll PZ1toD1 \gg & = & \texttt{PZ1} \ll D1toPZ1 \gg \ ; \\
\vdots & \vdots & \vdots \\
\ll PZmtoDm \gg & = & \texttt{PZm} \ll DmtoPZm \gg \ ; \\
\ll T1toD1 \gg & = & \texttt{T1} \ll D1toT1 \gg \ ; \\
\vdots & \vdots & \vdots \\
\ll \boldsymbol{TmtoDm} \gg & \boldsymbol{=} & \boldsymbol{Tm} \ll \boldsymbol{DmtoTm} \gg \ ;
\end{array} \\
\textsf{end } \texttt{MultiVPV}
\end{array}
$$

Abbildung 4.7.2: ANDL-Spezifikation für das verteilte Multiprozessorsystem `MultiVPV`

4.7.2 Der Warteraum für verteiltes Dispatching

Zu Beginn von Abschnitt 4.5 wurde beschrieben, um welche Funktionalitäten die Queue erweitert wird, um die Anforderungen der n Prozesse gleicher Priorität nach *einem* Prozessor zu behandeln. Im folgenden modellieren wir eine zentrale Queue, die die Schaltstelle zwischen n Prozessen und m Prozessoren bzw. deren Dispatchern bildet. Zum einen muß die Queue alle eintreffenden Prozessoranforderungen der Prozesse Pi berücksichtigen. Zum anderen muß sie auf alle vorliegenden Anfragen *Next* der m Dispatcher reagieren. Für die Behandlung der Anforderungen durch die Prozesse übernehmen wir die in Abschnitt 4.5.3 gezeigte Lösung. Für die Realisierung der Verbindungen zu den Dispatchern wird die Queue um folgende Funktionalitäten erweitert:

1. Die Queue reagiert auf jede Nachricht *Next* eines Dispatchers Dj.
2. Verfügt die Queue über mindestens so viele Prozessoranforderungen, wie von den Dispatchern angefordert werden, werden diese an die Dispatcher weitergegeben.

3. Verfügt die Queue über weniger Prozessoranforderungen, als die Dispatcher fordern, werden die verfügbaren Identifikatoren und zugehörigen Ports an die Dispatcher weitergegeben. An die übrigen Dispatcher wird die Nachricht *Empty* gesendet.

Am SSD von Abbildung 4.7.1 können wir ablesen, daß alle Pi für $i \in \{1, \dots, n\}$ durch einen Kanal $PitoQ$ mit Q verbunden sind. Zusätzlich ist die Queue mit jedem Dispatcher durch jeweils einen Kanal zu dem sie über das Schreib- bzw. das Leserecht verfügt, verbunden. Für die Spezifikation von Q gelten die in Tabelle 4.5.1 auf Seite 83 aufgeführten Nachrichtentypen. Hierbei ist zu beachten, daß für alle Kanäle $DjtoQ$ bzw. $QtoDj$ die dort für die Kanäle $DtoQ$ bzw. $QtoD$ festgelegten Typen definiert sind und über alle Kanäle Ports der Menge $N_{MultiVPV}$ gesendet werden dürfen.

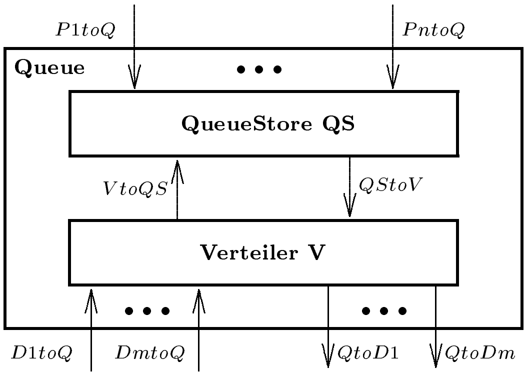


Abbildung 4.7.3: Die Queue für verteiltes Dispatching

Die oben gegebenen Erläuterungen zeigen, daß die Aufgaben der Queue in zwei Teile zerfallen: Die Behandlung der Prozesse und der Dispatcher. Um dieser Zweiteilung der Queue gerecht zu werden, nutzen wir die hierarchische Konzeption von Focus aus, durch die eine Komponente als verteiltes System modelliert werden kann. Abbildung 4.7.3 zeigt die Queue als verteiltes System, und in Abbildung 4.7.4 wird die Andl-Spezifikation Queue mit den Subkomponenten *QueueStore QS* und *Verteiler V* angegeben.

agent Q
input channels $P1toQ : S_{P1toQ}, \dots, PntoQ : S_{PntoQ},$
$D1toQ : S_{D1toQ}, \dots, DmtoQ : S_{DmtoQ}$
output channels $QtoD1 : S_{QtoD1}, \dots, QtoDm : S_{QtoDm}$
is network
$\ll QtoD1, \dots, QtoDm, VtoQS \gg \; = \; \mathsf{V} \ll D1toQ, \dots, DmtoQ, QStoV \gg;$
$\ll QStoV \gg \; = \; \mathsf{QS} \ll P1toQ, \dots, PntoQ, VtoQS \gg;$
end Q

Abbildung 4.7.4: Andl-Spezifikation der Queue bei verteiltem Dispatching

Die Nachrichtentypen für die neu eingeführten internen Kanäle $QStoV$ und $VtoQS$ werden in den folgenden Abschnitten festgelegt.

4.7.2.1 Der Verteiler

Die Komponente V hat die Aufgabe, alle Anforderungen *Next* durch alle Dispatcher aufzunehmen, eine entsprechende Anzahl von Identifikatoren mit zugehörigen Ports bei der Komponente QS anzufordern und diese an die Dispatcher weiterzuleiten. Liegen nicht genügend Anforderungen durch Prozesse vor, wird dies den Dispatchern mitgeteilt, damit diese *aktives Warten* realisieren können.

agent V
  input channels $D1toQ : S_{D1toQ}, \ldots, DmtoQ : S_{DmtoQ}, QStoV : S_{QStoV}$
  output channels $QtoD1 : S_{QtoD1}, \ldots, QtoDm : S_{QtoDm}, VtoQS : S_{VtoQS}$
  private channels $\emptyset$
is basic
  f_V $\boxed{\text{mit der Spezifikation von Seite 103}}$
end V

Abbildung 4.7.5: ANDL-Spezifikation von V

Im folgenden bezeichnen wir mit $PiList$ eine Liste von Prozeßidentifikatoren, mit $PortList$ eine Liste von Ports und mit d eine Liste zu Identifizierung der Dj.

(1) V wird mit einer Sequenz $\langle PiList, PortList \rangle$ über Kanal $QStoV$ gestartet. Gilt $\#PiList < m$, wird über die Kanäle $QtoDk$ mit $k \in \{\#PiList + 1, \ldots, m\}$ die Nachricht *Empty* gesendet und alle übrigen Anforderungen werden weitergeleitet.

(2) Erhält V über die Kanäle $DjtoQ$ die Nachricht *Next*, wird registriert, welche Dispatcher Anforderungen gesendet haben. Über Kanal $VtoQS$ wird eine entsprechende Anzahl von Identifikatoren angefordert.

(3) Erhält V über $QStoV$ eine Sequenz $\langle PiList, PortList \rangle$, sendet V jeweils ein $\langle Pi, !DtoPi \rangle$ über die Kanäle $QtoDj$ an die intern registrierten Dj. Für den Fall, daß $\#PiList < \#d$ gilt, wird an die Dj, deren Anforderung nicht erfüllt werden kann, die Nachricht *Empty* gesendet. Zudem werden alle von den Dispatchern gesendeten Nachrichten *Next* registriert. Über Kanal $VtoQS$ wird eine entsprechende Anzahl von Identifikatoren angefordert.

(4) Erhält V über Kanal $QStoV$ die Nachricht *Empty*, wird an alle intern registrierten Dj die Nachricht *Empty* gesendet. Zudem werden alle von den Dispatchern gesendeten Nachrichten *Next* registriert. Über Kanal $VtoQS$ wird eine entsprechende Anzahl von Identifikatoren angefordert.

Funktionsgleichungen für f_V

$\forall s \in \prod_{n \in N_{MultiVPV}} [S_n^*],\ j,l,t \in \{1,\dots,m\},\ in_j \in \{\langle Next\rangle, \langle\rangle\}, PiList \in \{P1,\dots,Pn\}^* \setminus \{\langle\rangle\},$
$PortList \in \{!DtoP1,\dots,!DtoPn\}^* \setminus \{\langle\rangle\},\ d,f \in \{1,\dots,m\}^*\ :$
$\exists h \in \{1,\dots,m\}^* \to Type_V\ :$

Für $\#PiList = m$:

(1a) $f_V(\{QStoV \mapsto \langle PiList, PortList\rangle\} \circ s)$
$= \{QtoD1 \mapsto \langle p_1, !Dto(p_1)\rangle, \dots, QtoDm \mapsto \langle p_l, !Dto(p_l)\rangle\} \circ h(\langle\rangle)(s)$

Für $\#PiList < m$:

(1b) $f_V(\{QStoV \mapsto \langle PiList, PortList\rangle\} \circ s)$
$= \{QtoD1 \mapsto \langle p_1, !Dto(p_1)\rangle, \dots, QtoDl \mapsto \langle p_l, !Dto(p_l)\rangle,$
$QtoD(l+1) \mapsto \langle Empty\rangle, \dots, QtoDm \mapsto \langle Empty\rangle\} \circ h(\langle\rangle)(s)$

(2) $h(\langle\rangle)(\{D1toQ \mapsto in_1, \dots, DmtoQ \mapsto in_m\} \circ s)\ =\ \{VtoQS \mapsto \langle \#(in_1 \circ \dots \circ in_m)\rangle\} \circ h(f)(s)$

Für $d \neq \langle\rangle$ und $\#PiList = \#d$:

(3a) $h(d)(\{QStoV \mapsto \langle PiList, PortList\rangle, D1toQ \mapsto in_1, \dots, DmtoQ \mapsto in_m\} \circ s)$
$= \{QtoDj_1 \mapsto \langle p_1, !Dtop_1\rangle, \dots QtoDj_t \mapsto \langle p_l, !Dtop_l\rangle,$
$VtoQS \mapsto \langle \#(in_1 \circ \dots \circ in_m)\rangle\} \circ h(f)(s)$

Für $d \neq \langle\rangle$ und $\#PiList < \#d$:

(3b) $h(d)(\{QStoV \mapsto \langle PiList, PortList\rangle, D1toQ \mapsto in_1, \dots, DmtoQ \mapsto in_m\} \circ s)$
$= \{QtoD1 \mapsto \langle p_1, !Dto(p_1)\rangle, \dots, QtoDl \mapsto \langle p_l, !Dto(p_l)\rangle,$
$QtoD(l+1) \mapsto \langle Empty\rangle, \dots, QtoD(\#d) \mapsto \langle Empty\rangle\}$
$VtoQS \mapsto \langle \#(in_1 \circ \dots \circ in_m)\rangle\} \circ h(f)(s)$

(4) $h(d)(\{QStoV \mapsto \langle Empty\rangle, D1toQ \mapsto in_1, \dots, DmtoQ \mapsto in_m\} \circ s)$
$= \{QtoDj_1 \mapsto \langle Empty\rangle, \dots QtoDj_t \mapsto \langle Empty\rangle, VtoQS \mapsto \langle \#(in_1 \circ \dots \circ in_m)\rangle\} \circ h(f)(s)$

wobei : $l = \#PiList$ und $j_1 = d[1], \dots, j_k = d[l], \dots, j_t = d[\#d]$
$p_1 = PiList[1], \dots, p_l = PiList[\#PiList]$
$(\#(\{k\} \copyright f) = 1 \ \Leftrightarrow\ in_k = \langle Next\rangle) \ \wedge\ (\#(\{k\} \copyright f) = 0 \ \Leftrightarrow\ in_k = \langle\rangle)$

4.7.2.2 Die Speichereinheit

Die Komponente QS übernimmt die Aufgaben der Queue, die durch die Schnittstelle zu den Prozessen charakterisiert ist. Sie muß alle eintreffenden Prozeßidentifikatoren und Ports abspeichern, darf keinen Prozeß bevorzugt behandeln und liefert dem Verteiler die geforderten Identifikatoren und Ports. Wir werden diese Komponente ausgehend von der Queue für das System mit n Prozessen modellieren. Hierbei behalten wir die Lösung für die Gleichbehandlung aller Prozesse bei. Das Verhalten wird so erweitert, daß Listen von Identifikatoren und Ports, entsprechend der von dem Verteiler geforderten Anzahl, weitergegeben werden. Abbildung 4.7.6 zeigt die ANDL-Spezifikation von QS.

agent QS
  input channels $P1toQ \; : \; S_{P1toQ}, \ldots, PntoQ \; : \; S_{PntoQ}, VtoQS \; : \; S_{VtoQS}$
  output channels $QStoV \; : \; S_{QStoV}$
  private channels $\emptyset$
is basic
  $\boldsymbol{f}_{\mathsf{QS}}$ $\boxed{\text{mit der Spezifikation von Seite 105}}$
end QS

Abbildung 4.7.6: ANDL-Spezifikation von QS

Die textuelle Beschreibung des Verhaltens lautet:

(1) QS wird durch den Erhalt von mindestens einem $\langle Pi, !DtoPi \rangle$ über die Kanäle $PitoQ$ gestartet.

  (a) Erhält QS $k \leq m$ Anforderungen, werden diese über Kanal $QStoV$ gesendet.

  (b) Erhält QS $k > m$ Anforderungen, wird eine Liste mit m Identifikatoren mit zugehörigen Ports über Kanal $QStoV$ gesendet. Die restlichen $k - m$ Identifikatoren werden gespeichert.

(2) QS erhält über Kanal $VtoQS$ ein $l \in I\!N$. p sei der interne Speicher von QS.

  (a) Gilt $\#p \geq l$, wird $\langle p[1] \circ \ldots \circ p[l] \rangle$ mit den zugehörigen Ports über Kanal $QStoV$ gesendet. Alle über die Kanäle $PitoQ$ eintreffenden Pi werden gespeichert.

  (b) Gilt $\#p < l$, werden die Liste p mit den zugehörigen Ports und möglicherweise über die Kanäle $PitoQ$ eintreffende $\langle Pi, !DtoPi \rangle$ über Kanal $QStoV$ gesendet. Alle über die Kanäle $PitoQ$ empfangenen Pi, die nicht bedient werden können, werden gespeichert.

(3) Gilt $p = \langle\rangle$ und erhält QS über alle Kanäle $PitoQ$ die Sequenz $\langle\rangle$, wird die Nachricht $Empty$ über Kanal $QStoV$ gesendet.

(4) Erhält QS über Kanal $VtoQS$ die Nachricht „0“, werden alle über die Kanäle $PitoQ$ eintreffenden Pi gespeichert.

Im Fall (1a) wird QS damit gestartet, daß es mindestens m Anforderungen durch Prozesse gibt. Da mit dem Start des Systems alle Prozessoren frei sind, können die Anforderungen der ersten m Prozesse bedient werden. In (1b) werden weniger als m Prozessoren benötigt.

Mit (2) sind anhand der Anforderung von k ausgewählten Prozessen drei Fallunterscheidungen zu betrachten. In (2a) liegen QS genügend Anforderungen im internen Speicher vor, die Prozesse und entsprechenden Ports werden an V weitergegeben. Die von weiteren Prozessen eintreffenden Anforderungen werden abgespeichert. In (2b) liegen intern keine k

Anforderungen vor. Mit (2b.1) und (2b.2) werden die Fälle unterschieden, in denen zum einen mindestens k und zum anderen weniger als k Prozesse einen Prozessor anfordern.

Funktionsgleichungen für f_{QS}

$$\forall s \in \prod_{n \in N_{MultiVPV}} [S_n^*],\ l,t \in \{1,\ldots,n\},\ k \in \{0,1,\ldots,m\},$$
$$in_t \in \{\langle Pt, !DtoPt\rangle, \langle\rangle\},\ p \in \{P1,\ldots,Pn\}^* \ :$$
$$\exists h \in (\{1,\ldots,n\} \times \{P1,\ldots,Pn\}^*) \to Type_{QS} \ :$$

Für $\#Init > m$ und $\bigvee_{i=1}^{n} in_i = \langle Pi, !DtoPi\rangle$:

$$(1a)\quad f_{QS}(\{P1toQ \mapsto in_1,\ldots,PntoQ \mapsto in_n\} \circ s) = \{QStoV \mapsto \langle Init[1] \circ \ldots \circ Init[m], !Dto(Init[1]),\ldots,!Dto(Init[m])\rangle\} \circ h(2, proc[m+1] \circ \ldots \circ proc[\#proc])(s)$$

Für $\#Init \leq m$ und $\bigvee_{i=1}^{n} (in_i = \langle Pi, !DtoPi\rangle)$:

$$(1b)\quad f_{QS}(\{P1toQ \mapsto in_1,\ldots,PntoQ \mapsto in_n\} \circ s) = \{QStoV \mapsto \langle Init, \{!DtoP1,\ldots,!DtoPn\}\copyright(in_1 \circ \ldots \circ in_n)\rangle\} \circ h(2,\langle\rangle)(s)$$

Für $\#p \geq k$ und $j = (l \bmod n)+1$:

$$(2a)\quad h(l,p)(\{P1toQ \mapsto in_1,\ldots,PntoQ \mapsto in_n, VtoQS \mapsto \langle k\rangle\} \circ s) = \{QStoV \mapsto \langle p[1],\ldots,p[k], !Dto(p[1]),\ldots,!Dto(p[k])\rangle\} \circ h(j, p[k+1] \circ \ldots \circ p[\#p] \circ proc)(s)$$

Für $\#p < k$ und $\#proc > (k - \#p)$:

$$(2b.1)\quad h(l,p)(\{P1toQ \mapsto in_1,\ldots,PntoQ \mapsto in_n, VtoQS \mapsto \langle k\rangle\} \circ s) = \{QStoV \mapsto \langle p, proc[1],\ldots,proc[k-\#p], !Dto(p[1]),\ldots,!Dto(p[\#p], !Dto(proc[1]),\ldots,!Dto(proc[k-\#p])\} \circ h(j, proc[(k-\#p)+1] \circ \ldots \circ proc[\#proc])(s)$$

Für $\#p < k$ und $\#proc \leq (k - \#p)$:

$$(2b.2)\quad h(l,p)(\{P1toQ \mapsto in_1,\ldots,PntoQ \mapsto in_n, VtoQS \mapsto \langle k\rangle\} \circ s) = \{QStoV \mapsto \langle p, proc[1],\ldots,proc[\#proc], !Dto(p[1]),\ldots,!Dto(p[\#p], !Dto(proc[1]),\ldots,!Dto(proc[\#proc])\} \circ h(j,\langle\rangle)(s)$$

$$(3)\quad h(l,\langle\rangle)(\{P1toQ \mapsto \langle\rangle,\ldots,PntoQ \mapsto \langle\rangle, VtoQS \mapsto \langle k\rangle\} \circ s) = \{QStoV \mapsto \langle Empty\rangle\} \circ h(j,\langle\rangle)(s)$$

$$(4)\quad h(l,p)(\{P1toQ \mapsto in_1,\ldots,PntoQ \mapsto in_n, VtoQS \mapsto \langle 0\rangle\} \circ s) = \{QStoV \mapsto \langle\rangle\} \circ h(j, p \circ proc)(s)$$

wobei : $Init = \{P1,\ldots,Pn\}\copyright(in_1 \circ \ldots \circ in_n)$
$proc = \{P1,\ldots,Pn\}\copyright(in_l \circ in_{(l \bmod n)+1} \circ \ldots \circ in_{((l+(n-2)) \bmod n)+1})$

Kapitel 5

Speicherverwaltung

Im vorliegenden Kapitel geben wir zunächst einen Überblick über die Aufgaben der Speicherverwaltung in Betriebssystemen mit Referenzen auf weiterführende Literatur. Wir erläutern den Anteil, der formal modelliert wird, und beschreiben und motivieren die Vorgehensweise bei der Erstellung der Modellierung. Die in Kapitel 4 erstellten Formalisierungen werden schrittweise erweitert und angepaßt, um die Konzepte der Speicherverwaltung in Verbindung mit der Ausführung einer Berechnung zu gewährleisten.

5.1 Einführung

Die Teile eines Rechensystems zur Eingabe und Aufbewahrung von Daten werden *Speicher* genannt. Ein Speicher besteht aus einer Anzahl von Speicherelementen, die alle eine gleiche, für den Speicher festgelegte Größe aufweisen. Diese sogenannten *Speicherzellen* werden durch numerische Bezeichner, ihre *Adressen*, eindeutig identifiziert. Im allgemeinen entspricht eine Speicherzelle einem *Byte*. Speicherzellen müssen zunächst lokalisiert werden, um sie anschließend zu nutzen; dieser Vorgang wird *Zugriff* genannt. Hierbei gibt es lesenden Zugriff (*Lesen*), bei dem Speicherinhalte abgefragt und nicht verändert werden, und schreibenden Zugriff (*Schreiben*), bei dem einer Speicherzelle ein neuer Wert zugewiesen wird. Eigenschaften, die die Speicher charakterisieren und durch die sie unterschieden werden, sind beispielsweise ihre *Kapazität*, die *Zugriffszeit* oder auch die *Zugriffsart*. Wir werden hier nicht weiter auf Möglichkeiten zur Klassifikation von Speichern eingehen, sondern verweisen u.a. auf die Kapitel 1.6 in [Sta92] oder 2 in [GS94].

Die wesentlichen Charakteristika eines Speichers sind dessen Zugriffszeit und Kapazität, die sich nicht beide gleichzeitig maximieren lassen. So ist die Zugriffszeit bei Speichern großer Kapazität meist langsam, während schnelle Speicher relativ teuer und verhältnismäßig klein sind. Durch die sogenannte *Speicherhierarchie* werden Speicher nach ihrer Zugriffszeit geordnet. Wir werden die Speicher, die in dieser Hierarchie an oberster Stelle stehen, als Haupt- oder *Arbeitsspeicher* (AS) und die weiteren Speicher als Platten- oder *Hintergrundspeicher* (HS) bezeichnen.

Prozesse benötigen Speicherplatz, vor allem während ihre Berechnungen in Verbindung mit dem Prozessor tatsächlich ausgeführt werden. Die Daten, die den aktuell auszuführenden Prozeß betreffen, müssen im Arbeitsspeicher, auf den der Prozessor direkt zugreift, verfügbar sein. Da die Kapazität des Arbeitsspeichers begrenzt ist, können i.a. nicht alle rechenbereiten Prozesse permanent im Arbeitsspeicher gehalten werden. Um die gesamte Datenhaltung zu gewährleisten, wird neben dem Arbeitsspeicher auch der Hintergrundspeicher genutzt. Der im Rechensystem verfügbare Speicherplatz wird aufgeteilt und von den Prozessen kooperativ genutzt. Die Maßnahmen, die hierfür erforderlich sind, übernimmt die *Speicherverwaltung* eines Betriebssystems.

Die Speicherverwaltung hat somit die Aufgabe, die freien und belegten Speicherbereiche sowohl im Arbeits- als auch im Hintergrundspeicher zu verwalten und Speicherbereiche an Prozesse zuzuweisen bzw. frei werdende Speicherbereiche entgegenzunehmen. Aktuell benötigte Daten sind im Arbeitsspeicher bereitzustellen und solche, die aktuell nicht benötigt werden, im Hintergrundspeicher abzulegen. Speicherverwaltungen werden darin unterschieden, ob Prozesse während ihrer Ausführung zwischen AS und HS transferiert bzw. ob bei Ausführung eines Prozesses alle oder nur Teile der Daten im Arbeitsspeicher gehalten werden. Eine weitere Aufgabe der Speicherverwaltung besteht darin, dafür Sorge zu tragen, daß freie Speicherbereiche passender Größe zur Verfügung stehen.

In diesem Kapitel werden wir uns mit den Maßnahmen beschäftigen, die erforderlich sind, um eine Berechnung durch einen Prozeß in Verbindung mit Prozessor *und Speicher* auszuführen. Dazu modellieren wir die Konzepte eines *virtuellen Speichers*, siehe zur Beschreibung Abschnitt 5.2.3. Die Belegung bzw. Freigabe von Speicherplatz bei der Erzeugung bzw. Terminierung eines Prozesses werden in Kapitel 7 behandelt.

5.2 Wesentliche Aspekte der Speicherverwaltung

Speicher bestehen aus Speicherzellen, die jeweils zur Speicherung eines Bytes (oder anderer Größen) geeignet sind. Auf den Zellen ist eine lineare Ordnung definiert, und jede Zelle wird durch ihre Adresse eindeutig identifiziert. Üblicherweise werden die Zellen nicht separat, sondern als Zusammenschluß zu *Regionen* verwaltet. Eine *Region* ist ein zusammenhängender Speicherbereich, der aus einer Menge von Zellen mit aufeinanderfolgenden Adressen besteht. Die Adresse einer Speicherzelle kann damit relativ zum Anfang einer Region bestimmt werden. Die Instruktionen, die auf dem Speicher ausgeführt werden, sind das *Lesen* bzw. *Schreiben* von Daten relativ zu einer Adresse. Die Adressen, die in einem Programm verwendet werden, sind i.a. symbolisch und stimmen nicht mit den *realen* Adressen im Speicher überein. Wir bezeichnen Adressen, die im Programm verwendet werden, als logisch oder *virtuell* und solche, die im Speicher tatsächlich auftreten, als *physikalisch.*

Sobald dokumentiert ist, in welcher Region ein Programm oder Daten gespeichert sind, kann eine physikalische Adresse entsprechend berechnet werden.

Zu den Aufgaben der Speicherverwaltung gehören die Dokumentation, ob eine Speicherzelle frei und ungenutzt oder belegt und genutzt ist, und das Buchführen über die freien Regionen. Die Speicherverwaltung muß gewährleisten, daß

1. die Anforderungen freier Regionen passender Länge erfüllt werden und

2. nicht mehr benötigte Regionen freigegeben und in den Speicher eingegliedert werden.

5.2.1 Speicherverwaltung ohne Paging

Bei diesem Konzept werden der gesamte Prozeß bei seiner Ausführung im Arbeitsspeicher gehalten und die Daten und Programme vollständig in einer zusammenhängenden Region gespeichert. Aktuell nicht benötigte Prozesse werden vom Arbeits- in den Hintergrundspeicher transferiert. Dieses Umspeichern der Prozesse wird *Swapping* genannt. Das Betriebssystem führt eine Liste freier Regionen. Wird für einen Prozeß Speicherplatz benötigt, so wird diese Liste nach einem Speicherbereich geeigneter Größe durchsucht. Die Auswahl eines freien Speicherbereichs wird nach unterschiedlichen Prinzipien durchgeführt. Genannt seien hier:

First-Fit: Die erste Region wird ausgewählt, die groß genug ist, um den Prozeß aufzunehmen.

Best-Fit: Es wird die Region ausgewählt, deren Größe die erforderliche Größe am wenigsten überschreitet.

Der Teil einer ausgewählten Region, der bei einer Belegung frei bleibt, wird wieder in die Liste der freien Speicherbereiche aufgenommen. Die Strategien verfügen über unterschiedliche Eigenschaften; siehe hierzu die entsprechende Literatur wie die Kapitel 3.2 in [Tan92], 8.4 in [GS94] oder 3.3 in [Fin88]. Bei dieser Vorgehensweise kann es vorkommen, daß viele Regionen geringer Größe entstehen. Es ist möglich, daß für die Abspeicherung eines Prozesses kein zusammenhängender Speicher passender Größe belegt werden kann, obwohl prinzipiell genügend freier Speicherplatz vorhanden ist, der sich allerdings aus vielen kleinen Stücken zusammensetzt. Dieses Problem wird *externe Fragmentierung* genannt; siehe die Kapitel 8.4.3 in [GS94], 5.2 in [Sta92] sowie 4.2.1 in [Spi95].

Um die Problematik der externen Fragmentierung abzuschwächen, können regelmäßig Maßnahmen zur Kompaktifizierung des Speichers durchgeführt werden: Freie Speicherbereiche werden zusammengefaßt, um freie Bereiche größerer Länge zu bilden. Prozesse müssen hierbei umgespeichert werden, was aufwendige Umrechnungen der Adressen notwendig macht; zur Beschreibung siehe beispielsweise Kapitel 8.4.3 von [GS94].

5.2.2 Speicherverwaltung mit Paging

Eine alternative Lösung zur Behebung des Problems der externen Fragmentierung besteht darin, einen Prozeß nicht mehr *am Stück* in *einer* Region abzuspeichern. Arbeits- und Hintergrundspeicher werden jeweils in Stücke gleicher Länge, sogenannte *Seitenrahmen* (oder Kacheln) bzw. *Blöcke*, aufgeteilt. Jeder Seitenrahmen bzw. Block kann über seine Anfangsadresse eindeutig identifiziert werden. Weiterhin wird der logische, also der von einem Prozeß benötigte Speicherbereich, ebenfalls in Stücke dieser Länge, sogenannte *Seiten*, aufgebrochen. Die Seiten eines Prozesses können nun entweder durch Seitenrahmen oder durch Blöcke realisiert werden, und ein Prozeß wird nicht mehr in einer zusammenhängenden Region, sondern verteilt auf die Seitenrahmen des Arbeitsspeichers gespeichert.

Die physikalische Adresse einer Instruktion wird aus deren logischer Adresse bestimmt. Hierfür muß die Anfangsadresse des Seitenrahmens bekannt sein. Durch die Speicherverwaltung wird darüber Buch geführt, ob ein Seitenrahmen belegt oder frei ist, und wo eine Seite gespeichert ist. Diese Informationen werden in sogenannten *Seitentabellen* gehalten; nachzulesen in den Kapiteln 8.5.1 und 8.5.2 in [GS94] oder 3.3.2 in [Tan94]. Die Speicherverwaltungseinheit, die eine logische Adresse in eine Speicheradresse abbildet, wird auch *Memory Management Unit* (MMU) genannt, siehe Abschnitt 8.2 in [GS94].

5.2.3 Virtueller Speicher

Mit dem in Abschnitt 5.2.2 beschriebenen Verfahren ist es noch immer notwendig, Prozesse zu ihrer Ausführung vollständig im Speicher zu halten. Dies hat zur Folge, daß die Größe des für einen Prozeß zur Verfügung stehenden Speicherplatzes durch die Größe des Arbeitsspeichers beschränkt ist. Speicherverwaltungen, die das Konzept des *virtuellen Speichers* realisieren, siehe die Kapitel 3.2 in [Tan94], 9 in [GS94] oder 5.2 in [Spi97a], sind dadurch charakterisiert, daß keineswegs immer die vollständige Beschreibung eines Prozesses im Arbeitsspeicher gehalten werden muß. Es reicht aus, die Seiten eines Prozesses, die für den aktuellen Stand der Berechnung benötigt werden, im Arbeitsspeicher zu realisieren. Damit ist es auch möglich, daß der von einem Prozeß benötigte Speicherplatz die Größe des Arbeitsspeichers überschreiten kann. Zudem benötigt ein Prozeß für seine Ausführung weniger physikalischen Speicherplatz, und eine größere Anzahl von Prozessen kann gleichzeitig im Arbeitsspeicher gehalten werden. Wir werden für unser System eine Speicherverwaltung mit virtuellem Speicher modellieren.

Der von einem Prozeß benötigte Speicherplatz wird in Seiten aufgeteilt, von denen nun ein Teil durch Seitenrahmen und ein anderer Teil durch Blöcke realisiert ist. Ein Prozeß ist somit *gemischt* AS- und HS-realisiert, und die Speicherverwaltung muß dokumentieren, *wo* eine Seite aktuell realisiert ist. Greift der Prozessor bei der Ausführung einer Instruktion auf eine Adresse zu, deren zugehörige Seite aktuell nicht AS-realisiert ist, erzeugt das Betriebssystem einen sogenannten *Seitenfehler*. Der Prozeß wird blockiert, und die Ausführung der Berechnung kann erst dann fortgesetzt werden, wenn die Seite im Arbeitsspeicher verfügbar ist. Das Laden einer Seite in einen Seitenrahmen wird *Einlagern* genannt. Möglicherweise ist es notwendig, zunächst einen Seitenrahmen frei zu machen, um eine benötigte Seite einlagern zu können. Das Entfernen einer Seite aus dem Arbeitsspeicher mit ihrer Abspeicherung in einem freien Block des Hintergrundspeichers wird *Auslagern* genannt. Die Entscheidung darüber, wann eine Seite ausgelagert wird, kann nach verschiedenen Kriterien getroffen werden, die mittels entsprechender *Seitenersetzungsalgorithmen* durchgesetzt werden; siehe Abschnitt 5.2.3.1. Zusätzlich müssen Verfahren bereitgestellt werden, die die Zuordnung von Seitenrahmen zu Prozessen gewährleisten; siehe Abschnitt 5.2.3.2.

5.2.3.1 Seitenersetzungsalgorithmen

Wenn ein Prozeß aktuell über den Prozessor verfügt, so führt der Prozessor die Instruktionen des Programms mit den entsprechenden Adressen aus. Hierbei werden logische Adressen in physikalische Adressen umgerechnet, und die Speicherverwaltung hat Kenntnis darüber, ob eine zu einer logischen Adresse gehörende Seite im Arbeitsspeicher realisiert ist, oder nicht. Sobald eine benötigte Seite nicht im AS-realisiert ist, erzeugt das Betriebssystem einen Seitenfehler, der Prozeß wird blockiert, und der Prozessor wird freigegeben. Die Speicherverwaltung muß Maßnahmen ergreifen, damit der Prozeß die Ausführung seiner Berechnung nach angemessener Zeit fortsetzen kann. Zu diesen Maßnahmen gehört insbesondere, daß die fehlende Seite in einen Seitenrahmen gespeichert wird. Hierbei sind zwei Fälle zu unterscheiden:

1. Seitenrahmen, die für den Prozeß genutzt werden können, sind frei. In diesem Fall ist die Seite in einen solchen einzulagern.

2. Alle Seitenrahmen, die ein Prozeß nutzen kann, sind bereits belegt. In diesem Fall muß zunächst ein Seitenrahmen frei gemacht werden, indem eine im AS-realisierte Seite in den Hintergrundspeicher gelegt (*verdrängt*) wird. Erst nach der Verdrängung kann die benötigte Seite eingelagert werden.

Die Auswahl einer Seite, die verdrängt werden kann, erfolgt über verschiedene Kriterien, die mittels sogenannter *Seitenersetzungsalgorithmen* durchgesetzt werden; siehe die Kapitel 3.4 in [Tan94], 6.4.2 in [Spi95] und 9.5 in [GS94]. Die Speicherverwaltung führt hierbei eine Liste der Seiten, die im Arbeitsspeicher realisiert sind. Die Auswahl einer Seite erfolgt beispielsweise mit einer festgelegten Wahrscheinlichkeit bei der *Random*-Strategie, oder, weil sie am längsten im Arbeitsspeicher realisiert ist, mit der *FIFO*-Strategie, oder gemäß weiterer Kriterien, die ausführlich in der oben genannten Literatur beschrieben sind. Derartige Algorithmen werden beispielsweise gemäß der *Anzahl* der erzeugten Seitenfehler bewertet. So gilt für FIFO die sogenannte *FIFO-Anomalie*, siehe u.a. Kapitel 3.5 in [Tan94], da die Anzahl der Seitenfehler mit der Steigerung der Anzahl der Seitenrahmen wächst. Bei der Entscheidung über eine Auslagerung sollte berücksichtigt werden, ob die Seite oft oder selten benutzt wird. Wir werden die Strategie LRU (*Least Recently Used*) modellieren, bei der die Seite ausgewählt wird, die am längsten *nicht* benutzt wurde; siehe Abschnitt 5.5.2.

5.2.3.2 Zuordnung von Seitenrahmen

Weitere Aufgaben der Speicherverwaltung bestehen darin, einem Prozeß eine ausreichende Anzahl von Seitenrahmen im AS zur Verfügung zu stellen, um zu vermeiden, daß permanent Seiten ein- und ausgelagert werden. Zudem ist festzulegen, nach welchen Kriterien Seiten in Seitenrahmen eingelagert werden; hierbei wird unterschieden zwischen *vorausplanenden* und *anfordernden* Verfahren. Entweder werden aufgrund einer Analyse der erforderlichen Speicherzugriffe Seiten bereits vor Ausführung des Prozesses im Arbeitsspeicher eingelagert oder die Seiten werden dann eingelagert, wenn sie benötigt werden, vergleiche *Working-Set* bzw. *Demand Paging* in Abschnitt 3.6 von [Tan94].

Die Varianten unterscheiden sich zudem darin, ob die Entscheidungen für eine Seite oder mehrere Seiten getroffen werden, und ob jeder Prozeß separat (*prozeßlokal*) behandelt wird oder alle koexistierenden Prozesse (*prozeßglobal*) berücksichtigt werden. Unsere Modellierung erfolgt gemäß einer reinen *lokalen Einzelseiten-Anforderungsstrategie*.

5.2.3.3 Bestimmung physikalischer Adressen

In diesem Abschnitt beschäftigen wir uns mit der Bestimmung physikalischer Adressen auf der Basis der Adreßräume von Arbeits- und Hintergrundspeicher sowie den virtuellen Adreßräumen der Prozesse. Diese Beschreibung ist bereits auf die in den folgenden Abschnitten gezeigten Modellierungen zugeschnitten; wir führen hier Bezeichner ein, die in den Spezifikationen verwendet werden.

Sowohl die virtuellen als auch die physikalischen Adressen werden durch numerische Bezeichner modelliert. Wir beschreiben Adreßräume durch die Mengen

$$\begin{aligned} ASA &= \{0, \ldots, |AS| - 1\} \quad \text{für den Arbeitsspeicher}^1 \\ HSA &= \{0, \ldots, |HS| - 1\} \quad \text{für den Hintergrundspeicher} \end{aligned}$$

sowie den virtuellen Adreßraum eines Prozesses Pi durch $VA_{Pi} = \{0, \ldots, |VA_{Pi}| - 1\}$. Hierbei gelte $\{|VA_{Pi}|, |AS|, |HS|\} \subseteq I\!N$.

Der virtuelle Adreßraum eines Prozesses sei jeweils in Seiten der Länge $l_p \in I\!N$ aufgeteilt. Damit erhalten wir die Menge der Seitenidentifikatoren für einen Prozeß Pi mit $Sid_{Pi} = \{0, \ldots, |VA_{Pi}| \texttt{ div } l_p\}$. Da die Größe der Seitenrahmen und Blöcke in Arbeits- und Hintergrundspeicher mit der einer Seite übereinstimmt, erhalten wir entsprechend

$$\begin{aligned} SRid &= \{0, \ldots, |AS| \texttt{ div } l_p\} \quad \text{die Menge der Seitenrahmenidentifikatoren} \\ Bid &= \{0, \ldots, |HS| \texttt{ div } l_p\} \quad \text{die Menge der Blockidentifikatoren} \end{aligned}$$

Die Berechnung der physikalischen Adressen erfolgt mittels sogenannter Speicherfunktionen, siehe auch die Abschnitte 5.2 in [Spi97a] und 6.4 in [Spi95]. Zu einer virtuellen Adresse

[1] Mit $|M|$ bezeichnen wir die Mächtigkeit einer Menge M.

va läßt sich der zugehörige Seitenidentifikator bestimmen durch va `div` l_p. Der Identifikator des Seitenrahmens, in dem die Seite realisiert ist, sei sr. Damit können wir die physikalische Adresse ra berechnen mit $ra = sr * l_p + va \texttt{ mod } l_p$.

Die Seiten sind, wie bereits beschrieben, die Datenobjekte, die einen Prozeß charakterisieren. Für das von uns betrachtete Systemverhalten ist ausschließlich die Modellierung der Verwaltungsmaßnahmen und nicht die Datenmodellierung von Interesse. Aus diesem Grund gehen wir stark abstrahierend davon aus, daß eine Seite p vom Typ $Pages$ ist.

5.3 Methodische Vorgehensweise

Die in Kapitel 4 entwickelte Spezifikation stellt das Kernstück unserer systematischen Entwicklung eines Systems dar, mit dem wesentliche Konzepte des Ressourcenmanagements auf hohem Abstraktionsniveau gewährleistet sind. Bei der in Kapitel 4 behandelten Aufgabenstellung hatten wir uns ausschließlich auf die Prozessorverwaltung beschränkt und sind davon ausgegangen, daß ein Prozeß zur Ausführung seiner Berechnung nur das Betriebsmittel *Prozessor* benötigt.

Ausgehend von dem in Abschnitt 4.5 beschriebenen und spezifizierten Einprozessorsystem werden wir nun unser System um Funktionalitäten und Konzepte einer Speicherverwaltung mit virtuellem Speicher, siehe Abschnitt 5.2.3, erweitern. Zur konsequenten Weiterführung unserer methodischen Vorgehensweise bei der schrittweisen und systematischen Entwicklung des letztendlich geforderten Systems sind folgende Punkte zu berücksichtigen:

- Das Kernsystem wird dahingehend untersucht, welche Komponenten
  - ohne Veränderung übernommen werden können;
  - in ihrem Verhalten angepaßt, erweitert und verändert werden müssen.
- Komponenten der Speicherverwaltung müssen konzipiert, neu spezifiziert und in die bestehende Modellierung integriert werden.
- Insgesamt werden die Spezifikationen in dem von uns gewählten Stil erstellt:
  - der Umfang bleibt überschaubar und die Modellierung nachvollziehbar;
  - spezielle Komponenten sind für spezifische Aufgaben zuständig;
  - Aufgaben, die *lokal* gehalten werden können, werden durch separate Komponenten und nicht von einer *Zentrale* bearbeitet.

Zur Funktionalität des Systems werden die Konzepte eines virtuellen Speichers mit Seiten für die Prozesse sowie Seitenrahmen und Blöcke zu deren Realisierung hinzugefügt. Jedem Prozeß wird eine feste Menge von Seitenrahmen zu seiner exklusiven Nutzung zugeteilt. Deren Zuteilung modellieren wir in Kapitel 7 im Rahmen der Prozeßerzeugung. Die zur Fortführung einer Berechnung benötigten Seiten werden eingelagert, wenn sie noch nicht im Arbeitsspeicher realisiert sind. Insgesamt ergibt sich folgende Aufgabenstellung:

Ein Einprozessorsystem mit Prozessorverwaltung wird um Aufgaben der Speicherverwaltung erweitert. Hierbei werden Konzepte eines virtuellen Speichers mit einer reinen prozeßlokalen Einzelseiten-Anforderungsstrategie modelliert.

Das System aus Abschnitt 4.5 besteht aus n Prozessen, den Komponenten zur Prozessorverwaltung – der Queue, dem Timer und dem Dispatcher – und dem Prozessor. Die Prozesse repräsentieren die Berechnungen der Benutzer, konkurrieren um den Prozessor und nutzen ihn gemeinsam. Mit den bereits genannten Aufgaben der Speicherverwaltung steht fest, daß das in Abschnitt 4.5 entwickelte System um die Komponenten *Arbeits-* und *Hintergrundspeicher* sowie einen *Speicherverwalter* erweitert wird. Mit dem Arbeitsspeicher stehen sowohl der Prozessor – zur Ausführung von Instruktionen – als auch der Speicherverwalter – zum Umspeichern der Seiten – in Verbindung. Der Hintergrundspeicher wird nur dem Speicherverwalter zugeordnet. Speicherverwalter und Prozesse müssen in einer noch zu präzisierenden Art und Weise direkt oder indirekt verbunden sein, da Prozesse in Abhängigkeit der verfügbaren Seiten blockiert und entblockiert werden. Aus diesen Überlegungen ergibt sich die in Abbildung 5.3.1 dargestellte Struktur des erweiterten Systems, wobei bereits modellierte Teilbereiche markiert sind. Im folgenden werden wir kurz erläutern, welche Erweiterungen und Anpassungen im Detail vorgenommen werden.

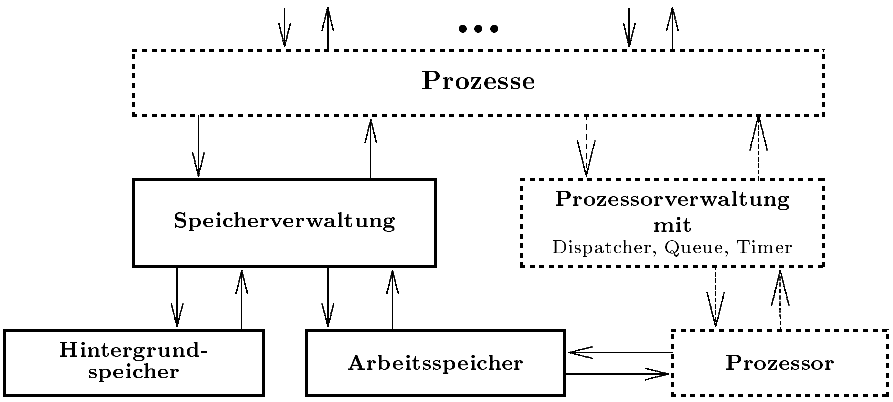


Abbildung 5.3.1: Erweitertes System zur Speicherverwaltung

Die Aufgaben des Speicherverwalters umfassen: Die Dokumentation der Belegung von Seitenrahmen und Blöcken, das Erkennen und Melden von Seitenfehlern, die Durchsetzung der LRU-Strategie sowie das Umspeichern der Seiten von Blöcken in Seitenrahmen und umgekehrt. Da wir ein prozeßlokales Verfahren modellieren, können die benötigten Informationen über die Seitenrahmen, also insbesondere bzgl. LRU und der Seitenfehler, für jeden Prozeß separat behandelt werden. Wir führen für jeden Prozeß einen prozeßlokalen Speicherverwalter ein, der über die Realisierung der Seiten des Prozesses und die ihm zugeordneten Seitenrahmen Kenntnis hat. Bei Bedarf wird die Aufgabe des Umspeicherns einer Seite an den globalen Speicherverwalter weitergeleitet. Insgesamt erhalten wir die Aufteilung des Speicherverwalters in einen globalen und entsprechend der Anzahl der Prozesse mehrere lokale Speicherverwalter. Für die bereits modellierten Komponenten gilt:

Prozesse: Bisher war zur Beschreibung einer Berechnung ausschließlich eine Folge von Instruktionen ausreichend. Diese abstrakte Darstellung eines Programms ist nun zu erweitern. Jedem Schritt wird eine Adresse zugeordnet, die den Speicherzugriff charakterisiert. Zusätzlich ist zu berücksichtigen, daß ein Prozeß in den neuen Zustand *waiting* übergeht, sobald ein Seitenfehler auftritt.

Prozessor: Speicherzugriffe erfolgen in Verbindung mit der Ausführung von Berechnungsschritten und werden somit durch den Prozessor ausgeführt. Die Informationen für die Berechnung der *physikalischen* Adressen stehen mit dem prozeßlokalen Speicherverwalter zur Verfügung. Analog zur Vorgehensweise bei der Zuteilung des Prozessors werden wir einen prozeßlokalen Speicherverwalter an den Prozessor weitergeben, wenn dem Prozeß der Prozessor zugewiesen wird.

Für die hier zu erstellenden Modellierungen können somit die Komponenten des Systems aus Abschnitt 4.5, die ausschließlich für die Durchführung der Prozessorverwaltung zuständig sind, also die Queue, der Dispatcher und der Timer, unverändert übernommen werden. Im folgenden passen wir die Modellierungen von Prozeß und Prozessor an und erweitern sie und spezifizieren die Komponenten *prozeßlokale* und *globaler Speicherverwalter* neu.

5.4 Das erweiterte System mit Speicherverwaltung

Mit der in Abschnitt 5.3 vorgegebenen Aufgabenstellung und den ersten Erläuterungen zur Modellierung ergibt sich ein verteiltes FOCUS-System `MemManag` mit n Prozessen und den Komponenten der Prozessorverwaltung, das um den *globalen* und n *lokale Speicherverwalter* sowie Arbeits- und Hintergrundspeicher ergänzt wird. Analog zur Modellierung der Zuteilung des Prozessors wird ein lokaler Speicherverwalter SV_{Pi} an den Prozessor gebunden, sobald Prozeß und Prozessor gekoppelt sind. Das System durchläuft, bezogen auf Prozeß Pi, die in Abbildung 5.4.1 gezeigten zwei Phasen: der assoziierte lokale Speicherverwalter ist entweder an den Prozeß oder an den Prozessor gebunden. Zur Steigerung der Übersichtlichkeit verzichten wir auf die graphische Darstellung der Komponenten, die die Verwaltung des Prozessors gewährleisten, und zeigen den strukturellen Aufbau nur für einen repräsentativen Prozeß Pi. Im modellierten System sind n Prozesse enthalten.

Ausgehend von diesem SSD erstellen wir nun, analog zu allen bisherigen Spezifikation, die in Abbildung 5.4.2 gezeigte ANDL-Spezifikation. Wir werden bei der Spezifikation dieses Netzes auf solche Kanäle und Komponenten verzichten, die in Kapitel 4 modelliert wurden und unverändert in die Erweiterung des Systems übernommen werden können.

Analog zu allen bisher erstellten ANDL-Spezifikationen wird die *initiale* Vernetzung des Systems `MemManag` angegeben. Wir gehen davon aus, daß initial alle Prozesse Pi mit ihrem privaten Speicherverwalter SV_{Pi} verbunden sind und der Prozessor keinem der Prozesse zugeteilt ist. Die SV_{Pi} sind mit dem globalen Speicherverwalter durch jeweils einen Kanal mit Schreib- und einen mit Lesezugriff verbunden. Die Nachrichtentypen für die neu

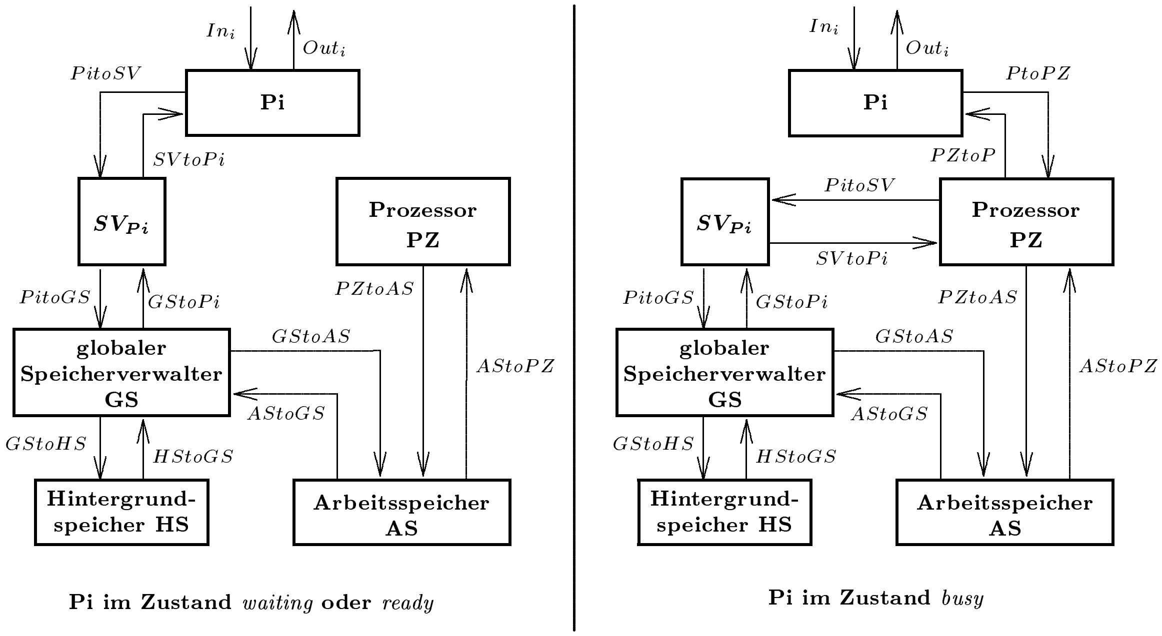


Abbildung 5.4.1: System `MemManag` für Prozeß Pi

agent `MemManag`

input channels $\ldots, In_i : S_{In_i}, \ldots$

output channels $\ldots, Out_i : S_{Out_i}, \ldots$

is network

$$
\begin{array}{lcl}
\vdots & \vdots & \quad\vdots \\
\ll Out_i, \boldsymbol{PitoSV}, \ldots \gg & = & \mathtt{Pi} \ll In_i, \boldsymbol{SVtoPi}, \ldots \gg \; ; \\
\ll PitoGS, SVtoPi \gg & = & \mathtt{SV}_{Pi} \ll GStoPi, PitoSV \gg \; ; \\
\ll \boldsymbol{PZtoAS}, \ldots \gg & = & \mathtt{PZ} \ll \boldsymbol{AStoPZ}, \ldots \gg \; ; \\
\vdots & \vdots & \quad\vdots \\
\ll GStoP1, \ldots, GStoPn, & & \\
\qquad GStoAS, GStoHS \gg & = & \mathtt{GS} \ll P1toGS, \ldots, PntoGS \\
 & & \qquad AStoGS, HStoGS \gg \; ; \\
\ll AStoGS, AStoPZ \gg & = & \mathtt{AS} \ll GStoAS, PZtoAS \gg \; ; \\
\ll HStoGS \gg & = & \mathtt{HS} \ll GStoHS \gg \; ;
\end{array}
$$

[*sowie die Komponenten: Dispatcher, Queue und Timer*]

end `MemManag`

Abbildung 5.4.2: ANDL-Spezifikation für das System mit Speicherverwaltung eingeführten Kanäle sind den folgenden Abschnitten zu entnehmen. Die für das System gültigen Kanalbezeichner werden durch die Menge $?!N_{Mem}$ definiert.

5.5 Prozeßlokale Speicherverwaltung

Aufgrund der Einführung in die Speicherverwaltungen in Abschnitt 5.2 setzen wir voraus, daß die allgemeine Aufgabenstellung bekannt ist. Wir haben festgelegt, daß eine prozeßlokaler Einzelseiten-Anforderungsstrategie modelliert wird. Damit ist es möglich, die einem Prozeß lokal zugeordneten Aufgaben durch eine prozeßlokale Komponente zu modellieren.

Zur Nutzung des Arbeitsspeichers wird einem Prozeß bei seiner Erzeugung eine festgelegte Menge von Seitenrahmen zur exklusiven Nutzung zugewiesen, siehe hierzu auch Kapitel 7. Die lokale Speicherverwaltung muß für die Ausführung eines Prozesses und die Realisierung der Seiten des Prozesses lediglich über diese Seitenrahmen Buch führen. Eine benötigte Seite ist ausschließlich mit diesen Seitenrahmen realisiert bzw. kann nur in diese eingelagert werden. Zur Berechnung einer physikalischen Adresse muß dokumentiert sein, *wo* eine Seite realisiert ist, auf die zugegriffen werden soll. Der lokale Speicherverwalter führt die Umrechnung der Adressen durch. Es ist zu beachten, daß wir eine *reine* Anforderungsstrategie modellieren, bei der zu Beginn keine Seite des Prozesses im Arbeitsspeicher realisiert ist. Eine weitere Aufgabe besteht darin, den Seitenersetzungsalgorithmus zu spezifizieren. Auch hierfür reicht es aus, die dem Prozeß zugeordneten Seiten zu berücksichtigen. Der lokale Speicherverwalter dokumentiert die Zugriffe auf die Seiten und führt die für LRU geforderten Maßnahmen durch.

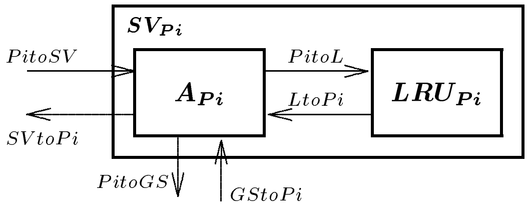


Abbildung 5.5.1: Der private Speicherverwalter SV_{Pi}

Gemäß der so beschriebenen Aufgabenteilung wird ein lokaler Speicherverwalter als Netz modelliert, siehe das SSD in Abbildung 5.5.1. Jedes SV_{Pi} besteht aus einer Komponente LRU_{Pi} und einer Komponente A_{Pi}, die Adressen umrechnet und Seitenersetzungen veranlaßt. In den folgenden Abschnitten werden beide Unterkomponenten modelliert.

5.5.1 Lokale Seitenverwaltung und Adreßberechnung

Dem in Abbildung 5.5.1 gezeigten SSD entnehmen wir die Schnittstelle der Komponente A_{Pi}. Über den Kanal *PitoSV* empfängt sie virtuelle Adressen, zu denen sie entweder die entsprechende physikalische Adresse oder einen Seitenfehler melden muß. A_{Pi} ist mit dem globalen Speicherverwalter verbunden, der bei Bedarf das Umspeichern der Seiten zwischen Arbeits- und Hintergrundspeicher übernimmt. Über die Kanäle $PitoGS$ und $GStoPi$ werden Seitenidentifikatoren bzw. die Informationen über eine erfolgreich durchgeführte Umspeicherung gesendet. Mit der LRU-Komponente ist A_{Pi} über die Kanäle $PitoL$ und $LtoPi$ zur Weiterleitung von Seitenidentifikatoren verbunden. Mit diesen Identifikatoren wird zum einen die LRU-Strategie realisiert und zum anderen überprüft, ob die Seite im Arbeitsspeicher verfügbar ist. Falls eine Seite nicht AS-realisiert ist, unterscheiden wir:

- Alle dem Prozeß zugeordneten Seitenrahmen sind belegt. A_{Pi} erhält den Identifikator der Seite, die gemäß LRU auszulagern ist.
- Es gibt noch freie, dem Prozeß zugeordnete Seitenrahmen. A_{Pi} wird informiert, und es muß keine Seite ausgelagert werden.

Mit Abbildung 5.5.2 erhalten wir die ANDL-Spezifikation von Komponente A_{Pi}.

agent A_{Pi}
  input channels $PitoSV : S_{PitoSV}, LtoPi : S_{LtoPi}, GStoPi : S_{GStoPi}$
  output channels $SVtoPi : S_{SVtoPi}, PitoL : S_{PitoL}, PitoGS : S_{PitoGS}$
  private channels $\emptyset$
is basic
  $\boldsymbol{f_{A_{Pi}}}$ $\boxed{\text{mit der Spezifikation von Seite 121}}$
end A_{Pi}

Abbildung 5.5.2: ANDL-Spezifikation von A_{Pi}

Die in Abschnitt 5.4 festgelegte Menge N_{Mem} der Kanalbezeichner des modellierten Systems `MemManag` wird um die Kanalbezeichner $\{PitoL, LtoPi \mid \forall i \in \{1, \ldots, n\}\}$ erweitert.

A_{Pi} empfängt virtuelle Adressen $va \in VA_{Pi}$ und kann auf der Basis der in Abschnitt 5.2.3.3 erklärten Funktionen sowohl die zugehörige Seite als auch die zugeordnete physikalische Adresse bestimmen. Wir setzen voraus, daß A_{Pi} in der Lage ist, einem Seitenidentifikator den Identifikator des Seitenrahmens bzw. Blockes, in dem die Seite aktuell realisiert ist, zuzuordnen. Als Reaktion auf eine über Kanal $PitoSV$ empfangene virtuelle Adresse va meldet die Komponente A_{Pi} an den Prozessor die Nachricht

- $MemOk$ und die physikalische Adresse ra, falls die Seite AS-realisiert ist;
- $PageFault$, falls die Seite aktuell nicht im AS-realisiert ist.

Ist die benötigte Seite nicht AS-realisiert, so müssen Maßnahmen zur Behebung des Seitenfehlers eingeleitet werden. Zu diesem Zweck sendet A_{Pi} über Kanal $PitoGS$ an den globalen Speicherverwalter die Nachricht

- $SwapIn(p, b, sr)$ zur Umspeicherung der Seite p aus Block b in den Seitenrahmen sr;

- $SwapInOut((p,b),(q,sr))$, wobei der Seitenrahmen sr frei gemacht werden muß, um die aktuell in Block b realisierte, und nun benötigte Seite p dort einzulagern. Die Seite q wird im nun freien Block b realisiert.

Kanal n	Nachrichtenmengen S_n	
$PitoSV$	VA_{Pi}	
$SVtoPi$	$ASA \cup \{PageFault, MemOk\}$	
$PitoL$	$SRid^* \cup \{Access(p) \mid p \in Sid_{Pi}\}$	
$LtoPi$	$\{MemOk\} \cup \{(PageFault, FreeSR(sr)) \mid sr \in SRid\}$	
	$\cup \{(PageFault, Swap(p)) \mid p \in Sid_{Pi}\}$	$\cup\ ?!N_{Mem}$
$PitoGS$	$\{(Pi, SwapIn(p,b,sr) \mid p \in Sid_{Pi}, sr \in SRid, b \in Bid\}$	
	$\cup \{(Pi, SwapInOut((q,b),(p,sr)) \mid$	
	$p,q \in Sid_{Pi}, b \in Bid, sr \in SRid\}$	
$GStoPi$	$\{ASIn(p,sr), HSIn(p,b) \mid p \in Sid_{Pi}, sr \in SRid, b \in Bid\}$	
	$\cup \{(k, blist) \mid k \in I\!N, blist \in Bid^*\}$	

Tabelle 5.5.1: Nachrichtentypen für die prozeßlokalen Komponenten A_{Pi}

Vom globalen Speicherverwalter empfängt A_{Pi} über Kanal $GStoPi$ Nachrichten der Form $ASIn(p,sr)$ und $HSIn(p,b)$, durch die gemeldet wird, in welchem Seitenrahmen sr bzw. Block b eine Seite p realisiert ist. Tabelle 5.5.1 enthält die den Kanälen von A_{Pi} zugeordneten Nachrichtentypen. Die Bedeutung der bisher nicht erklärten Nachrichten wird mit den folgenden Spezifikationen von A_{Pi} und LRU oder bei der Modellierung des globalen Speicherverwalters in Abschnitt 5.6 deutlich.

Wir gehen davon aus, daß jede Komponente A_{Pi} über einen internen Zustand verfügt, in dem alle Informationen über die Realisierung der Seiten in Seitenrahmen bzw. Blöcken enthalten sind. Dieser Zustand wird bei der Erzeugung von A_{Pi} mit den Blockidentifikatoren initialisiert, die die *physikalische* Realisierung der Seiten des Prozesses bei seiner Erzeugung beschreiben. Diesen internen Zustand modellieren wir durch ein *array*, auf dessen Elemente durch die Seitenidentifikatoren eines Prozesses Pi zugegriffen werden kann. Es gelte $n_{Pi} = |Sid_{Pi}| - 1$.

$$State_{A_{Pi}} \ : \ \texttt{array}\ [\ 0 : n_{Pi}\ ]\ \texttt{of}\ (SRid\ \cup\ Bid)$$

Für die hier gegebene textuelle Beschreibung seien $z : State_{A_{Pi}}$ und l_p die Größe einer Seite $p \in Pages$. Auf ein Element i des `arrays` greifen wir mit $z[i]$ zu. Zu einer gegebenen virtuellen Adresse $va \in VA_{Pi}$ kann der zugehörige Seitenidentifikator bestimmt werden; es gelte $p_{va} = va\ \texttt{div}\ l_p$. Mit diesen Festlegungen und Erklärungen geben wir das für den privaten Speicherverwalter geforderte Verhalten in der üblichen Form an.

(1) A_{Pi} wird durch den Empfang der Nachricht $(k, blist)$ über Kanal $GStoPi$ gestartet. Der interne Zustand wird mit der Realisierung von Pi im HS initialisiert.

A_{Pi} wird mit der HS-Realisierung des Prozesses initialisiert. Diese umfaßt die Anzahl der Seiten $k = |Sid_{Pi}|$ und eine Liste *blist* von Identifikatoren der Blöcke, in denen die Seiten des Prozesses realisiert sind. In der Spezifikation des globalen Speicherverwalters GS in Abschnitt 5.6 wird das Versenden dieser Nachricht über Kanal $GStoPi$ noch nicht enthalten sein. Diese Initialisierung wird erst bei der Erzeugung eines Prozesses in Abschnitt 7.7.2 in Kapitel 7 behandelt.

(2) Empfängt A_{Pi} über Kanal $PitoSV$ eine virtuelle Adresse va, werden die Nachricht $Access(p_{va})$ über Kanal $PitoL$ gesendet und va abgespeichert; es wird überprüft, ob die zu va gehörige Seite AS-realisiert ist.

(3) Erhält A_{Pi} die Nachricht $MemOk$, wird die Nachricht (va, ra) über Kanal $SVtoPi$ gesendet. Hierbei wird die physikalische Adresse ra gemäß der Speicherfunktion bestimmt. Es gilt $ra \; = \; z[p_{va}] * l_p + va \texttt{ mod } l_p$.

Die zu va gehörende Seite ist AS-realisiert, es tritt kein Seitenfehler auf und die physikalische Adresse wird bekannt gegeben.

(4) Erhält A_{Pi} über Kanal $LtoPi$ die Nachricht $PageFault$, wird diese über Kanal $SVtoPi$ weitergeleitet. Zudem werden folgende Fälle unterschieden:

(a) Wenn A_{Pi} die Nachrichten $PageFault$ und $FreeSR(k)$ erhält, wird die Nachricht $(Pi, SwapIn(p_{va}, z[p_{va}], k))$ über Kanal $PitoGS$ gesendet. A_{Pi} wartet auf die erfolgreiche Einlagerung der Seite.

Der Seitenrahmen $k \in SRid$ ist noch frei, und die Seite p_{va} kann dort eingelagert werden. Sie muß vom Block $z[p_{va}]$ in k umgespeichert werden.

(b) Erhält A_{Pi} die Nachrichten $PageFault$ und $Swap(q)$, wird die Nachricht

$$(Pi, SwapInOut((p_{va}, z[p_{va}]), (q, z[q])))$$

über Kanal $PitoGS$ gesendet. A_{Pi} wartet auf das erfolgreiche Umspeichern dieser Seiten.

Es gibt keine freien Seitenrahmen mehr. Die Seite q, die aktuell im Seitenrahmen $z[q]$ realisiert ist, muß ausgelagert werden. A_{Pi} veranlaßt, die Umspeicherung der Seiten q und p_{va}.

(5) Erhält A_{Pi} über Kanal $GStoPi$ die Nachricht $ASIn(p, sr)$, werden sr in $z[p]$ abgespeichert und die Nachricht $MemOk$ über Kanal $SVtoPi$ gesendet.

Die Seite p wurde erfolgreich im Seitenrahmen sr gespeichert. sr wird registriert und Pi wird entblockiert.

(6) Erhält A_{Pi} über Kanal $GStoPi$ die Nachrichten $ASIn(p, sr)$ und $HSIn(q, b)$, werden sr in $z[p]$ und b in $z[q]$ abgespeichert und die Nachricht $MemOk$ über Kanal $SVtoPi$ gesendet.

Die Seiten p und q wurden umgespeichert. Die Identifikatoren des Seitenrahmen bzw. Blockes werden abgespeichert und Pi entblockiert.

Wir erhalten die folgende formale Spezifikation.

Funktionsgleichungen für $f_{A_{Pi}}$

$\forall s \in \prod_{n \in N_{Mem}} [S_n^*],\ z \in State_{A_{Pi}},\ va \in VA_{Pi},\ p, q \in Sid_{Pi},\ sr \in SRid,\ b \in Bid,\ blist \in Bid^* :$
$\exists g \in State_{A_{Pi}} \ \to\ Type_{A_{Pi}},$
$\quad h_1 \in (State_{A_{Pi}} \times VA_{Pi}) \ \to\ Type_{A_{Pi}},\ h_2 \in (State_{A_{Pi}} \times Sid_{Pi}) \ \to\ Type_{A_{Pi}} \ :$

(1) $f_{A_{Pi}}(\{GStoPi \mapsto \langle (k, blist) \rangle\} \circ s) \ = \ g(z)(s)$

wobei $z[k] = blist[k+1]$ für $0 \leq k \leq n_{Pi}$

(2) $g(z)(\{PitoSV \mapsto \langle va \rangle\} \circ s) \ = \ \{PitoL \mapsto \langle Access(p_{va}) \rangle\} \circ h_1(z, va)(s)$

(3) $h_1(z, va)(\{LtoPi \mapsto \langle MemOk \rangle\} \circ s) \ = \ \{SVtoPi \mapsto \langle va, z[p_{va}] * l_p + va \ \mathtt{mod} \ l_p \rangle\} \circ g(z)(s)$

(4a) $h_1(z, va)(\{LtoPi \mapsto \langle PageFault, FreeSR(k) \rangle\} \circ s)$
$\quad = \ \{SVtoPi \mapsto \langle PageFault \rangle,$
$\quad\quad PitoGS \mapsto \langle (Pi, SwapIn(p_{va}, z[p_{va}], k)) \rangle\} \circ h_2(z, p_{va})(s)$

(4b) $h_1(z, va)(\{LtoPi \mapsto \langle PageFault, Swap(q) \rangle\} \circ s)$
$\quad = \ \{SVtoPi \mapsto \langle PageFault \rangle,$
$\quad\quad PitoGS \mapsto \langle (Pi, SwapInOut((p_{va}, z[p_{va}]), (q, z[q]))) \rangle\} \circ h_2(z, p_{va})(s)$

(5) $h_2(z, p)(\{GStoPi \mapsto \langle ASIn(p, sr) \rangle\} \circ s) \ = \ \{SVtoPi \mapsto \langle MemOk \rangle\} \circ g(z[p] := sr)(s)$

(6) $h_2(z, p)(\{GStoPi \mapsto \langle ASIn(p, sr), HSIn(q, b) \rangle\} \circ s)$
$\quad = \ \{SVtoPi \mapsto \langle MemOk \rangle\} \circ g(z[p] := sr, z[q] := b)(s)$

wobei $p_{va} = va \ \mathtt{div} \ l_p$ mit $l_p = |p|$ für $p \in Sid_{Pi}$

5.5.2 Die Modellierung von LRU

In Abschnitt 5.2.3.1 wurde die allgemeine Funktionsweise der Seitenersetzungsalgorithmen beschrieben. Wir werden das Verfahren LRU modellieren, bei dem die Seite ausgewählt wird, auf die am längsten *nicht* zugegriffen wurde. Die von der Speicherverwaltung geführte Liste der Seiten muß so verwaltet werden, daß sie Informationen über Zugriffe zu den Seiten enthält. Dieser Algorithmus wird i.a. durch verkettete Listen oder spezielle Hardware realisiert. Wir werden im folgenden eine Modellierung in Focus mit einer endlichen Sequenz, die die Seitenbezeichner enthält, angeben.

LRU mittels endlicher Sequenzen

Die Dokumentation der prozeßlokalen Seitenzugriffe erfolgt auf der Basis einer endlichen Sequenz von Seitenidentifikatoren. In einer Sequenz $pst \in Sid_{Pi}^*$ werden alle Seiten registriert, die aktuell AS-realisiert sind. Betrachten wir den Fall, daß auf eine Seite $page \in Sid_{Pi}$ zugegriffen wird. Die Sequenz ist so geordnet, daß $ft.pst$ die Seite identifiziert, auf die am längsten *nicht* zugegriffen wurde. Das letzte Element von pst identifiziert somit die Seite, auf die als letztes zugegriffen wurde. Für die Modellierung der LRU-Strategie gilt:

1. $page$ ist in pst enthalten: $page$ wird aus pst entfernt und an die Sequenz angefügt, da auf die Seite zugegriffen wird.
2. $page$ ist nicht in pst enthalten: Es tritt ein Seitenfehler auf. Zu dessen Behebung muß eine Seite gemäß LRU ausgelagert werden. Diese Seite wird durch $ft.pst$ identifiziert.

Basierend auf der in Abschnitt 2.2 erklärten Filterfunktion © definieren wir die Hilfsfunktion $\odot$, mit der ein Element aus einer endlichen Sequenz entfernt wird. Für eine beliebige Nachrichtenmenge M sowie $st \in M^*$ und $a \in M$ gilt: $\{a\} \odot st \;=\; (M \setminus \{a\})$ © st

Mit $\odot$ können wir die beiden oben genannten Fälle formalisieren:

1. Gilt $\#(\{page\}$©$pst) = 1$, erhalten wir: $pst' \;=\; (\{page\} \odot pst) \circ page$
2. Gilt $\#(\{page\}$©$pst) = 0$, erhalten wir: $pst' \;=\; rt.pst.$

Damit haben wir eine Möglichkeit erklärt, LRU mittels endlicher Sequenzen zu modellieren.

Ausgehend von dem in Abbildung 5.5.1 gezeigten SSD geben wir die in Abbildung 5.5.3 gezeigte ANDL-Spezifikation an. Die den Kanälen zugeordneten Nachrichtentypen wurden bereits in Tabelle 5.5.1 festgelegt.

agent LRU_{Pi}
  input channels $PitoL \;:\; S_{PitoL}$
  output channels $LtoPi \;:\; S_{LtoPi}$
  private channels $\emptyset$
is basic
    $\boldsymbol{f_{LRU_{Pi}}}$ $\boxed{\text{mit der Spezifikation von Seite 123}}$
end LRU_{Pi}

Abbildung 5.5.3: ANDL-Spezifikation von LRU_{Pi}

Mit diesen Erläuterungen erarbeiten wir nun die Spezifikation der Komponente LRU_{Pi}. In Abschnitt 5.5 wurde erklärt, daß LRU_{Pi} für die Durchführung des LRU-Verfahrens bzgl. der Seiten des Prozesses und für die Verwaltung der dem Prozeß zugeordneten Seitenrahmen zuständig ist. Zur Charakterisierung des Verhaltens unterscheiden wir die Fälle:

1. Die geforderte Seite ist AS-realisiert. Es wird gemeldet, daß der Speicherzugriff erfolgen kann. Die Dokumentation der Seitenzugriffe wird aktualisiert.
2. Die geforderte Seite ist nicht eingelagert, es wird ein Seitenfehler gemeldet. Zusätzlich wird entweder eine Seite, die ausgelagert werden kann, oder ein noch freier Seitenrahmen bekanntgegeben.

Für die Buchführung über die dem Prozeß Pi zugeordneten Seitenrahmen verfüge LRU_{Pi} über eine weitere Liste $kst \in SRid^*$. Für das Verhalten von LRU_{Pi} gilt:

(1) LRU_{Pi} wird mit dem Empfang einer Sequenz $kInit$ gestartet und geht in den Zustand $(kInit, \langle\rangle)$ über.

(2) LRU_{Pi} erhält über Kanal $PitoL$ eine Nachricht $Access(page)$, und es gilt

$$\#(\{page\}\copyright pst) = 1$$

Über Kanal $LtoPi$ wird die Nachricht $MemOk$ gesendet, und LRU_{Pi} geht in den Zustand $(\{page\} \odot pst) \circ page$ über.

(3) LRU_{Pi} erhält über Kanal $PitoL$ eine Nachricht $Access(page)$, und es gilt

$$\#(\{page\}\copyright pst) = 0$$

Über Kanal $LtoPi$ wird die Nachrichten $PageFault$ gesendet und zusätzlich

- **(a)** $Swap(ft.pst)$ bei $kst = \langle\rangle$. LRU_{Pi} geht in den Zustand $(rt.pst) \circ page$ über.

  Es gibt keine freien Seitenrahmen mehr, eine Seite muß ausgelagert werden. Es wird registriert, daß auf die Seite zugegriffen wird.

- **(b)** $FreeSR(ft.kst)$ bei $kst \neq \langle\rangle$. LRU_{Pi} geht in den Zustand $pst \circ page$ über.

  Es gibt noch freie prozeßlokale Seitenrahmen, so daß keine Seite ausgelagert werden muß. Es wird registriert, daß auf die Seite zugegriffen wird.

Diese textuelle Beschreibung wird in folgende formale Spezifikation umgesetzt.

Funktionsgleichungen für f_{LRU}

$\forall s \in \prod_{n \in N_{Mem}} [S_n^*],\ page \in Sid_{Pi},\ pst \in Sid_{Pi}^*,\ k \in SRid,\ kst, kInit \in SRid^*$:
$\exists\ h \in (SRid^* \times Sid_{Pi}^*) \ \rightarrow\ Type_{LRU_{Pi}}$:

(1) $f_{LRU_{Pi}}(\{PitoL \mapsto \langle kInit\rangle\} \circ s) \ =\ \{LtoPi \mapsto \langle\rangle\} \circ h(kInit, \langle\rangle)(s)$

Für $\#(\{page\}\copyright pst) = 1$:

(2) $h(kst, pst)(\{PitoL \mapsto \langle Access(page)\rangle\} \circ s)$
$= \{LtoPi \mapsto \langle MemOk\rangle\} \circ h(kst, (\{page\} \odot pst) \circ page)(s)$

Für $\#(\{page\}\copyright pst) = 0$:

(3a) $h(\langle\rangle, pst)(\{PitoL \mapsto \langle Access(page)\rangle\} \circ s)$
$= \{LtoPi \mapsto \langle PageFault, Swap(ft.pst)\rangle\} \circ h(\langle\rangle, ((rt.pst) \circ page) \circ page)(s)$

Für $kst \neq \langle\rangle$ und $\#(\{page\}\copyright pst) = 0$:

(3b) $h(kst, pst)(\{PitoL \mapsto \langle Access(page)\rangle\} \circ s)$
$= \{LtoPi \mapsto \langle PageFault, FreeSR(ft.kst)\rangle\} \circ h(kst, pst \circ page)(s)$

5.6 Der globale Speicherverwalter

Der globale Speicherverwalter GS bildet die Schnittstelle zwischen den lokalen Speicherverwaltern SV_{Pi} sowie den Speichern AS und HS und ist für das Umspeichern von Seiten zuständig. GS empfängt von allen Komponenten SV_{Pi} die Nachrichten $SwapIn$ und $SwapOut$, die nicht verloren gehen dürfen, veranlaßt die Umspeicherungen und meldet deren Erfolg mit den entsprechenden Informationen an die lokalen SV_{Pi}.

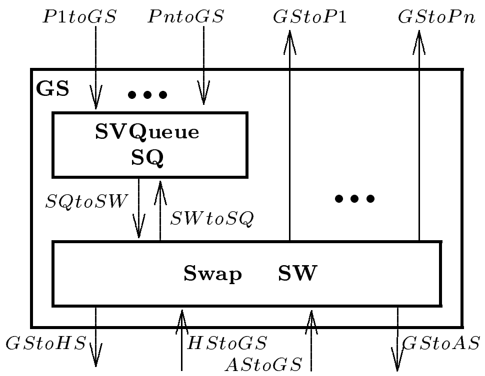


Abbildung 5.6.1: Der globale Speicherverwalter GS

Entsprechend zur Modellierung des Dispatchers zur Verwaltung von n Prozessoren in Abschnitt 4.6.3 werden wir hier eine konzeptuell ähnliche Modellierung vornehmen. Der globale Speicherverwalter ist aufgeteilt in

1. eine Queue SQ, die alle Nachrichten der SV_{Pi} entgegennimmt, und
2. eine Komponente SW, die die Ausführung der Umspeicherungen veranlaßt.

Das in Abbildung 5.6.1 für die Komponente GS gezeigte SSD wird in die in Abbildung 5.6.2 vorgestellte ANDL-Spezifikation umgesetzt.

agent GS
  input channels $P1toGS : S_{P1toGS}, \ldots, PntoGS : S_{PntoGS},$
  $AStoGS : S_{AStoGS}, HStoGS : S_{HStoGS}$
  output channels $GStoP1 : S_{GStoPn}, \ldots, GStoPn : S_{GStoPn},$
  $GStoAS : S_{GStoAS}, GStoHS : S_{GStoHS}$
is network
  $\ll SQtoSW \gg \;=\; \mathtt{SQ} \ll SWtoSQ, P1toGS, \ldots, PntoGS \gg$;
  $\ll SWtoSQ, GStoAS, GStoHS,$
  $GStoP1, \ldots, GStoPn \gg \;=\; \mathtt{SW} \ll SQtoSW, AStoGS, HStoGS \gg$;
end GS

Abbildung 5.6.2: ANDL-Spezifikation für den globalen Speicherverwalter GS

5.6.1 Die Queue des globalen Speicherverwalters

Die hier zu modellierende Queue ist dafür zuständig, alle von den lokalen SV_{Pi} eintreffenden Aufträge zum Umspeichern von Seiten entgegenzunehmen und diese auf Anfrage an die Komponente SW weiterzugeben. Zunächst geben wir, ausgehend von Abbildung 5.6.1, die ANDL-Spezifikation für SQ an, siehe Abbildung 5.6.3.

agent SQ

  input channels $SWtoSQ : S_{SWtoSQ}, P1toGS : S_{P1toGS}, \ldots, PntoGS : S_{PntoGS}$
  output channels $SQtoSW : S_{SQtoSW}$
  private channels $\emptyset$

is basic

  f_{SQ} $\boxed{\text{mit der Spezifikation von Seite 126}}$

end SQ

Abbildung 5.6.3: ANDL-Spezifikation von SQ

Für die Kanäle sind die in Tabelle 5.6.1 definierten Nachrichtentypen gültig. Alle Nachrichtentypen zu den hier nicht angegebenen Kanäle können den weiteren Abschnitten dieses Kapitels entnommen werden. Die Spezifikation von SQ erfolgt analog zu der in Abschnitt 4.3.7 gezeigten Modellierung mit den Anpassungen an die hier gültige Schnittstelle.

Kanal n	Nachrichtenmengen S_n
$SWtoSQ$	$\{Next\} \cup ?!N_{Mem}$
$SQtoSW$	$\{Empty\} \cup \bigcup_{i=1}^{n} S_{PitoGS} \cup ?!N_{Mem}$

Tabelle 5.6.1: Nachrichtentypen für SW

Zur Beschreibung des Verhaltens von SQ gilt:

(1) Alle über die Kanäle $PitoGS$ eintreffenden $SwapIn$- und $SwapOut$-Nachrichten sind entgegenzunehmen und zu speichern. Die erste Nachricht wird initial über den Kanal $SQtoSW$ ausgegeben.

(2) Erhält SQ über Kanal $SWtoSQ$ die Nachricht $Next$ und ist der interne Speicher str aktuell nicht leer, wird über Kanal $SQtoSW$ die Nachricht $ft.str$ gesendet.

(3) Ist der interne Speicher str aktuell leer und erhält SQ über Kanal $SWtoSQ$ die Nachricht $Next$, wird unterschieden:

  (a) Über mindestens einen Kanal $PitoGS$ wird eine Swap-Nachricht gesendet. SQ sendet über Kanal $SQtoSW$ eine Swap-Nachricht.

(b) Über keinen der Kanäle $PitoGS$ wird eine Swap-Nachricht empfangen. SQ sendet die Nachricht $Empty$ über Kanal $SQtoSW$.

Im Gegensatz zu den in Kapitel 4 modellierten Queues werden die eintreffenden Nachrichten stets in der Reihenfolge $1, 2, \ldots, n$ gespeichert. Dies bedeutet, daß immer die Nachrichten von Kanal $P1toGS$ zuerst und die von Kanal $PntoGS$ zuletzt abgearbeitet werden.

Basierend auf dem informell beschriebenen Verhalten erstellen wir die formale Spezifikation für SQ. Wir bezeichnen die Menge der Swap-Nachrichten mit $SwapMsg$, es gilt :

$$SwapMsg = \{(Pk, SwapIn(p, b, sr)), (Pk, SwapInOut((p, b), (q, sr)) \mid p, q \in Sid_{Pi},\ sr \in SRid, b \in Bid\}$$

Funktionsgleichungen für f_{SQ}

$\forall s \in \prod_{n \in N_{Mem}} [S_n^*],\ \ in_k \in \{\langle\rangle, \langle sw \rangle \mid sw \in SwapMsg\},\ \ str \in SwapMsg^*$:
$\exists\ h \in (SwapMsg^*) \rightarrow Type_{SQ}$:

Es gibt ein $k \in \{1, \ldots n\}$ mit $in_k \neq \langle\rangle$:

(1) $f_{SQ}(\{P1toGS \mapsto \langle in_1 \rangle, \ldots, PntoGS \mapsto \langle in_n \rangle\} \circ s)$
$= \{SQtoSW \mapsto \langle ft.(in_1 \circ \ldots \circ in_n) \rangle\} \circ h(rt.(in_1 \circ \ldots \circ in_n))(s)$

Für $str \neq \langle\rangle$:

(2) $h(str)(\{P1toGS \mapsto \langle in_1 \rangle, \ldots, PntoGS \mapsto \langle in_n \rangle, SWtoSQ \mapsto \langle Next \rangle\} \circ s)$
$= \{SQtoSW \mapsto \langle ft.str \rangle\} \circ h(rt.str \circ in_1 \circ \ldots \circ in_n)(s)$

Es gibt ein $k \in \{1, \ldots n\}$ mit $in_k \neq \langle\rangle$:

(3a) $h(\langle\rangle)(\{P1toGS \mapsto \langle in_1 \rangle, \ldots, PntoGS \mapsto \langle in_n \rangle, SWtoSQ \mapsto \langle Next \rangle\} \circ s)$
$= \{SQtoSW \mapsto \langle ft.(in_1 \circ \ldots \circ in_n) \rangle\} \circ h(rt.(in_1 \circ \ldots \circ in_n))(s)$

(3b) $h(\langle\rangle)(\{P1toGS \mapsto \langle\rangle, \ldots, PntoGS \mapsto \langle\rangle, SWtoSQ \mapsto \langle Next \rangle\} \circ s)$
$= \{SQtoSW \mapsto \langle Empty \rangle\} \circ h(\langle\rangle)(s)$

5.6.2 Die Swap-Komponente

Die Komponente SW ist dafür zuständig, die Umspeicherung von Seiten zu veranlassen und den Erfolg zu melden. Die lokalen SV_{Pi} senden die entsprechenden Aufträge mit den Informationen bzgl. Seiten-, Seitenrahmen- und Blockidentifikator an GS. Dort werden sie in SQ abgespeichert, aus der sie von SW entnommen werden. Ausgehend von Abbildung 5.6.1 geben wir in Abbildung 5.6.4 die ANDL-Spezifikation von SW an.

Für die Kanäle sind die mit Tabelle 5.6.2 festgelegten Nachrichtentypen gültig. Die Nachrichtentypen für die nicht aufgeführten Kanäle können den weiteren Abschnitten dieses Kapitels entnommen werden.

agent SW
  input channels $SQtoSW : S_{SQtoSW}, AStoGS : S_{AStoGS}, HStoGS : S_{HStoGS}$
  output channels $SWtoSQ : S_{SWtoSQ}, GStoP1 : S_{GStoP1}, \ldots, GStoPn : S_{GStoPn}$
  $GStoAS : S_{GStoAS}, GStoHS : S_{GStoHS}$
  private channels $\emptyset$
is basic
  f_{SW} mit der Spezifikation von Seite 129
end SW

Abbildung 5.6.4: ANDL-Spezifikation von SW

Kanal n	Nachrichtenmengen S_n
$GStoHS$	$\{HSRead(b) \mid b \in Bid\} \cup \{HSWrite(sr, pp) \mid b \in Bid, pp \in Pages\}$
$GStoAS$	$\{ASRead(sr) \mid sr \in SRid\} \cup \{ASWrite(sr, pp) \mid sr \in SRid, pp \in Pages\}$
$HStoGS$	$\{(b, pp) \mid b \in Bid, pp \in Pages\}$
$AStoGS$	$\{(sr, pp) \mid sr \in SRid, pp \in Pages\}$
	(alle obigen Mengen jeweils) $\cup\ ?!N_{Mem}$

Tabelle 5.6.2: Nachrichtentypen für SW

Bei der Beschreibung des Verhaltens sind die folgenden Fälle zu unterscheiden:

1. Eine Seite ist einzulagern. Dieser Auftrag wird durch $(Pi, SwapIn(p, b, sr))$ beschrieben. Darin enthalten sind die Identifikatoren b des Blockes, mit dem die Seite p aktuell realisiert ist, sowie sr des Seitenrahmens, in dem p realisiert werden soll. SW sorgt dafür, daß p aus HS entnommen und in AS durch sr realisiert wird.
2. Es sind sowohl eine Seite q ein- als auch eine Seite p auszulagern. Dieser Auftrag wird durch $(Pi, SwapInOut((p, b), (q, sr))$ beschrieben. Der Seitenrahmen sr, in dem q realisiert ist, muß frei gemacht werden. Das Umspeichern ist abgeschlossen, wenn q in HS im Block b und p in AS im Seitenrahmen sr realisiert sind.

Das Verhalten der Komponente SW beschreiben wir textuell mit:

(1) SW wird durch eine Nachricht $(pid, SwapIn(p, b, sr))$ über Kanal $SQtoSW$ gestartet. SW sendet die Nachricht $HSRead(b)$ über Kanal $GStoHS$ und geht in den Zustand $(pid, SwapIn(p, b, sr))$ über.

Für den Prozeß pid ist noch ein Seitenrahmen frei, die im Block b realisierte Seite muß im Seitenrahmen sr realisiert werden.

(2) Empfängt SW im Zustand $(pid, SwapIn(p, b, sr))$ über Kanal $HStoGS$ die Nachricht (b, hs), werden über Kanal $SVtoAS$ die Nachricht $ASWrite(sr, hs)$, über

Kanal $GSto(pid)$ die Nachricht $ASIn(p, sr)$ und über Kanal $SWtoSQ$ die Nachricht $Next$ gesendet.

Die Seite p wurde aus dem HS entnommen. Sie wird mit sr realisiert und die Erfolgsmeldung wird an SW gesendet.

(3) Erhält SW die Nachricht $(pid, SwapInOut((p, b), (q, sr)))$ über Kanal $SQtoSW$, sendet SW die Nachricht $HSRead(b)$ bzw. $ASRead(sr)$ über Kanal $GStoHS$ bzw. $GStoAS$ und geht in den Zustand $(pid, SwapInOut((p, b), (q, sr)))$ über.

Bevor die Seite p realisiert werden kann, muß ein Seitenrahmen frei gemacht werden. SW sendet an AS *und* an HS entsprechende Nachrichten.

(4) SW wartet auf die Erfolgsmeldungen von AS und HS, d.h. SW ist im Zustand $(pid, SwapInOut((p, b), (q, sr)))$. Wir unterscheiden:

(a) SW erhält über Kanal $AStoGS$ die Nachricht (sr, as) *und* über Kanal $HStoGS$ die Nachricht (b, hs). SW sendet folgende Nachrichten:

- $ASIn(p, sr)$ und $HSIn(q, b)$ über Kanal $GSto(pid)$,
- $ASWrite(sr, hs)$ über Kanal $GStoAS$,
- $HSWrite(b, as)$ über Kanal $GStoHS$ und
- $Next$ über Kanal $SWtoSQ$

(b) SW erhält über Kanal $AStoGS$ die Nachricht (sr, as) und geht in den Zustand $((pid, SwapInOut((p, b), (q, sr))), as)$ über.

(c) SW erhält über Kanal $HStoGS$ die Nachricht (b, hs) und geht in den Zustand $((pid, SwapInOut((p, b), (q, sr))), hs)$ über.

(5) SW ist im Zustand $(pid, SwapInOut((p, b), (q, sr))), hs)$ und erhält über Kanal $AStoGS$ die Nachricht (sr, as). Die Ausgabe erfolgt gemäß Punkt (4).

(6) SW ist im Zustand $(pid, SwapInOut((p, b), (q, sr))), as)$ und erhält die Nachricht (b, hs) über Kanal $HStoGS$. Die Ausgabe erfolgt gemäß Punkt (4).

(7) Erhält SW über Kanal $SQtoSW$ die Nachricht $Empty$, sendet SW die Nachricht $Next$ über Kanal $SWtoSQ$.

Die Punkte (4), (5) und (6) charakterisieren das Warten auf die erfolgreiche Umspeicherung einer Seite p von AS in HS und einer Seite q von HS in den mit der Auslagerung von p frei gewordenen Seitenrahmen. Diese können in einem Zeitintervall gemeinsam eintreffen, siehe (4a), oder nacheinander, siehe (4b) bzw. (4c), wobei jeweils auf die Erfolgsmeldung des anderen Speichers gewartet werden muß. Entsprechend dazu, welche Meldung bei SW zuerst eingetroffen ist, wird das weitere Verhalten beschrieben: durch (5), falls der Seitenrahmen zuerst frei wurde, und durch (6), falls der Block zuerst frei gemacht wurde.

Die textuelle Beschreibung wird in die formale Spezifikation umgesetzt. Wir verwenden die bereits in Abschnitt 5.6.1 definierte Menge $SwapMsg$ und

$$State_{SW} \ = \ Pid \ \times \ SwapMsg \ \times \ \bigcup_{i=1}^{n} Pages_{Pi}$$

Funktionsgleichungen für f_{SW}

$\forall s \in \prod_{n \in N_{Mem}} [S_n^*], \quad p, q \in \bigcup_{i=1}^{n} Sid_{Pi}, \quad sr \in SRid, \quad b \in Bid, \quad as, hs \in Pages \ :$
$\exists \ g \in (Pid \times SwapMsg) \ \rightarrow \ Type_{SW} \ , \ h_1, h_2 \in State_{SW} \ \rightarrow \ Type_{SW} \ ,$

(1) $f_{SW}(\{SQtoSW \mapsto \langle (pid, SwapIn(p,b,sr)) \rangle\} \circ s)$
$= \{GStoHS \mapsto \langle HSRead(b) \rangle\} \circ g(pid, SwapIn(p,b,sr))(s)$

(2) $g(pid, SwapIn(p,b,sr))(\{HStoGS \mapsto \langle (b,hs) \rangle\} \circ s)$
$= \{GStoAS \mapsto \langle ASWrite(sr,hs) \rangle,$
$GSto(pid) \mapsto \langle ASIn(p,sr) \rangle, SWtoSQ \mapsto \langle Next \rangle\} \circ f_{SW}(s)$

(3) $f_{SW}(\{SQtoSW \mapsto \langle (pid, SwapInOut((p,b),(q,sr))) \rangle\} \circ s)$
$= \{GStoHS \mapsto \langle HSRead(b) \rangle, GStoAS \mapsto \langle ASRead(sr) \rangle\}$
$\circ \, g(pid, SwapInOut((p,b),(q,sr)))(s)$

(4a) $g(pid, SwapInOut((p,b),(q,sr)))(\{AStoGS \mapsto \langle (sr,as) \rangle, HStoGS \mapsto \langle (b,hs) \rangle\} \circ s)$
$= \{GStoAS \mapsto \langle ASWrite(sr,hs) \rangle, GStoHS \mapsto \langle HSWrite(b,as) \rangle,$
$GSto(pid) \mapsto \langle ASIn(p,sr), HSIn(q,sr) \rangle, SWtoSQ \mapsto \langle Next \rangle\} \circ f_{SW}(s)$

(4b) $g(pid, SwapInOut((p,b),(q,sr)))(\{AStoGS \mapsto \langle (sr,as) \rangle\} \circ s)$
$= h_1(pid, SwapInOut((p,b),(q,sr)), as)(s)$

(4c) $g(pid, SwapInOut((p,b),(q,sr)))(\{HStoGS \mapsto \langle (b,hs) \rangle\} \circ s)$
$= h_2(pid, SwapInOut((p,b),(q,sr)), hs)(s)$

(5) $h_2(pid, SwapInOut((p,b),(q,sr)), hs)(\{AStoGS \mapsto \langle (sr,as) \rangle\} \circ s)$
$=$ *„siehe Ausgaben in Punkt (4a)“*

(6) $h_1(pid, SwapInOut((p,b),(q,sr)), as)(\{HStoGS \mapsto \langle (b,hs) \rangle\} \circ s)$
$=$ *„siehe Ausgaben in Punkt (4a)“*

(7) $f_{SW}(\{SQtoSW \mapsto \langle Empty \rangle\} \circ s) \ = \ \{SWtoSQ \mapsto \langle Next \rangle\} \circ f_{SW}(s)$

5.7 Arbeits- und Hintergrundspeicher

In unseren Modellierungen gehen wir davon aus, daß alle Speicheroperationen auf den Speichern des Rechensystem für die Prozesse interferenzfrei ablaufen. Konflikte, die bedingt durch die Hardware auftreten können, oder Algorithmen, die auf Datenstrukturen arbeiten, werden nicht modelliert. Der globale Speicherverwalter aus Abschnitt 5.6 und der Prozessor aus Abschnitt 5.8.2 sind die Komponenten, die in dem von uns spezifizierten System die Verbindung zu den Speichern bilden. Ausgehend von deren Modellierung gibt es in unserem System folgende Operationen auf den Speichern:

- Der Prozessor führt Speicheroperationen auf dem Arbeitsspeicher aus. Diese werden abstrakt beschrieben durch $(Step_i, ra_i)$. Da wir keine Berechnungsergebnisse erwarten, unterscheiden wir nicht zwischen Lese- und Schreiboperationen. Die Aufgabe des Arbeitsspeichers besteht darin, diese Nachricht entgegenzunehmen.
- Der globale Speicherverwalter sendet Nachrichten der Form $ASRead$ und $HSRead$ sowie $ASWrite$ und $HSWrite$ an Arbeits- und Hintergrundspeicher. Diese Nachrichten enthalten Seitenrahmen- bzw. Blockidentifikatoren sowie abzuspeichernde Seiten. Die Speicher geben den Inhalt des betroffenen Speicherbereichs aus oder belegen einen Speicherbereich mit den erhaltenen Daten. Auf die tatsächliche Ausführung der Lese- und Schreiboperationen gehen wir nicht ein.

Von beiden Speichern erwarten wir keine weiteren Funktionalitäten, so daß wir keine Details der Hardware modellieren werden. Daher verzichten wir auf die formale Spezifikation von Komponenten, die auf die oben beschriebenen Nachrichten reagieren.

5.8 Erweiterung von Prozeß und Prozessor

Mit den Modellierungen in den Abschnitten 5.5 und 5.6 wurde unser System um solche Komponenten erweitert, die für die Gewährleistung der Aufgaben der Speicherverwaltung nötig sind. Das System ist nun in der Lage, virtuelle Adressen in physikalische umzurechnen, Seitenzugriffe zu lokalisieren, Seitenfehler festzustellen und zu beheben. Im Einprozessorsystem mit n Prozessen aus Abschnitt 4.5, das die Basis für unsere Modellierungen bildet, haben wir Adressen bisher nicht berücksichtigt. Dies war beabsichtigt, da wir uns bei der Erarbeitung des Kernsystems ausschließlich um den Aspekt *Prozessorverwaltung* und die konkurrierende, gemeinsame Nutzung des Prozessors durch die Prozesse konzentriert haben. Im folgenden werden wir nun die Komponenten des Kernsystems aus Kapitel 4 so erweitern, daß auf Adressen Bezug genommen werden kann. Diese Erweiterungen haben Modifikationen der Spezifikationen der *Prozesse* und des *Prozessors* zur Folge, sie haben jedoch keine Auswirkungen auf die Komponenten *Timer*, *Dispatcher* und die *Queue*, da diese von der Durchführung der Berechnung, die mit einem Prozeß ausgeführt wird, nicht betroffen sind.

5.8.1 Prozesse mit Speicherfähigkeit

Die Zuordnung des lokalen Speicherverwalters SV_{Pi} zum zugehörigen Prozeß Pi erfolgt bei der Erzeugung eines Prozesses, siehe Kapitel 7. Daher werden wir auf diesen Aspekt in den folgenden Abschnitten noch nicht eingehen. Der lokale Speicherverwalter ist, wie wir an der Modellierung in Abschnitt 5.5.1 bereits gesehen haben, für die Realisierung der für den Prozeß relevanten Speicherverwaltungsaufgaben zuständig. Das in Kapitel 4 verwendete Zustandsdiagramm für Prozesse wird nun bzgl. der Speicherverwaltung erweitert. Diese Erweiterung enthält die Blockierung eines Prozesses, wenn ein Seitenfehler aufgetreten ist. Wir erhalten das in Abbildung 5.8.1 gezeigte erweiterte Zustandsdiagramm für Prozesse.

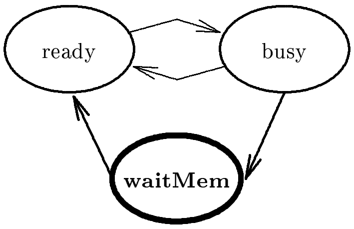


Abbildung 5.8.1: Zustandsdiagramm für Prozesse mit Speicherfähigkeit

Bisher wurde die Berechnung, für deren Ausführung ein Prozeß Pi zuständig ist, durch eine Sequenz $Step_1, \ldots, Step_n$ modelliert. Dabei sind wir davon ausgegangen, daß jedes $Step_i$ eine Instruktion repräsentiert, die vom Prozessor direkt ausgeführt werden kann. Wir erweitern diese abstrakte Beschreibung einer Berechnung dadurch, daß jedem Berechnungsschritt eine virtuelle Adresse zugeordnet wird. Folglich erhalten wir zur Modellierung einer Berechnung eine Sequenz von Paaren $(Step_1, va_1), \ldots, (Step_n, va_n)$ mit $va_i \in VA_{Pi}$. Zur Ausführung eines Berechnungsschrittes erhält der Prozessor nun ein $Step_i$ mit der zugehörigen virtuellen Adresse va_i. Pi erhält vom Prozessor entweder die Meldung, daß der Berechnungsschritt erfolgreich durchgeführt wurde, oder daß ein Seitenfehler aufgetreten ist, wodurch Pi blockiert wird.

agent Pi
  input channels $In_i : S_{In_i}, \boldsymbol{SVtoPi} : \boldsymbol{S_{SVtoPi}}, \ldots$
  output channels $Out_i : S_{Out_i}, \boldsymbol{PitoSV} : \boldsymbol{S_{PitoSV}}, \ldots$
  private channels $DtoPi : S_{DtoPi}$
is basic
  $\boldsymbol{f_{Pi}}$ $\boxed{\text{mit der Spezifikation von Seite 57 mit den Erweiterungen von Seite 133}}$
end Pi

Abbildung 5.8.2: ANDL-Spezifikation von Pi mit Verbindung zu SV_{Pi}

Der Prozessor benötigt für die Ausführung eines $Step_i$ die zu einer virtuellen Adresse gehörende physikalische Adresse sowie die Informationen über die realisierten Seiten. Er muß somit über die Informationen verfügen können, die vom lokalen Speicherverwalter verwaltet werden. Zu diesem Zweck gibt ein Prozeß Pi in unserer Modellierung seinen lokalen SV_{Pi} an den Prozessor weiter, sobald ihm dieser zugeteilt ist. Pi erhält den privaten Speicherverwalter zurück, wenn ihm der Prozessor entzogen wird. Für einen Prozeß ändert sich die Schnittstelle. Das SSD für die beiden Phasen eines Prozesses können wir in Abbildung 5.4.1 ablesen. Die in Abbildung 5.8.2 gezeigte ANDL-Spezifikation eines Prozesses Pi berücksichtigt nur die Erweiterung der Schnittstelle, die sich aus der Hinzunahme der Speicherverwaltung ergibt. Für die Kanäle gelten die in Tabelle 5.8.1 festgelegten Nachrichtentypen. Die hier nicht aufgeführten Kanäle wurden bereits in Kapitel 4 definiert und können unverändert übernommen werden. Die den Kanälen $PitoSV$ und $SVtoPi$ zugeordneten Nachrichtentypen können in Tabelle 5.5.1 nachgelesen werden. Gemäß unserer bisher gezeigten Vorgehensweise werden alle Erweiterungen und Anpassungen **markiert**.

Kanal n	Nachrichtenmengen S_n
$PtoPZ$	$(STEP \times \boldsymbol{VA})$
$PZtoP$	$\{Ok(Step_i) \mid Step_i \in STEP\}$
$\vdots$	$\vdots$
	$\cup\ ?!N_{Mem}$ (für alle obigen Zeilen)

Tabelle 5.8.1: Nachrichtentypen für den erweiterten Prozeß

Für das erweiterte Verhalten eines Prozesses fordern wir:

(1') Erhält Pi im Zustand *ready* über Kanal $DtoPi$ die Ports $?PZtoP$ und $!PtoPZ$, sendet Pi über Kanal $PtoPZ$ die nächste Instruktion **sowie die Ports $?\boldsymbol{SVtoPi}$ und $!\boldsymbol{PitoSV}$** und geht in den Zustand *busy* über.

(4') Erhält Pi im Zustand *busy* über Kanal $PZtoP$ die Sequenz

$$\langle Ok(ft.p), Suspend, \boldsymbol{?SVtoPi}, \boldsymbol{!PitoSV} \rangle,$$

werden die Verbindungen zum Prozessor gelöscht. Pi bewirbt sich erneut um den Prozessor und geht in den Zustand *ready* über.

(6) Erhält Pi im Zustand *busy* über Kanal $PZtoP$ die Nachricht $PageFault$ sowie die Ports $?SVtoPi$ und $!PitoSV$, wird die Verbindung zum Prozessor gelöst, und Pi geht in den Zustand *waitMem* über.

(7) Erhält Pi im Zustand *waitMem* über Kanal $SVtoPi$ die Nachricht $MemOk$, meldet er sich beim Dispatcher an und geht in den Zustand *ready* über.

Von den in der Spezifikation von $P1$ in Abschnitt 4.3.3 gezeigten Gleichungen haben wir hier die Punkte (1) und (4) aufgegriffen. In (1') und (4') kommt das Versenden von Ports – zur Weitergabe des lokalen Speicherverwalters an den Prozessor – explizit hinzu. Alle weiteren Schritte in Abschnitt 4.3.3 bleiben in ihrer Notation fast unverändert. Hierbei ist zu berücksichtigen, daß die Beschreibung der Berechnungen um virtuelle Adressen erweitert wurde. Diese Anpassung tritt in den Spezifikationen jedoch nur indirekt auf, daher verzichten wir auf ihre Darstellung.

Die Gleichungen (6) und (7) sind zur Spezifikation vollständig neu hinzugekommen. Sie beschreiben das Verhalten eines Prozesses, wenn ein Seitenfehler auftritt. In diesem Fall wird der Prozessor freigegeben und der Prozeß geht in den Zustand *waitMem* über. Er verläßt diesen Zustand, sobald die Ursachen des Seitenfehlers behoben wurden. In diesem Fall ist er bereit, seine Berechnung fortzuführen, und bewirbt sich wieder um den Prozessor.

Der Zustandsraum $State_{P1}$ wird um den Zustand $waitMem$ erweitert, und wir erhalten die neue Zustandsmenge $State'_{Pi}$ mit

$$State'_{Pi} \;=\; \{ready, busy, \boldsymbol{waitMem}\}$$

Die oben erstellte textuelle Beschreibung wird wie folgt umgesetzt:

Funktionsgleichungen für f_{Pi}

$\forall s \in \prod_{n \in N_{Mem}} [S_n^*],\;\; p \in (STEP \times \boldsymbol{VA_{Pi}})^* \;:$
$\exists\, h \in (\boldsymbol{State'_{Pi}} \times (\boldsymbol{STEP} \times \boldsymbol{VA_{Pi}})^*) \;\rightarrow\; Type_{Pi} \;:$

(1′) $h(ready, p)(\{DtoPi \mapsto \langle ?PZtoP, !PtoPZ\rangle\} \circ s)$
$\quad = \{PtoPZ \mapsto \langle ft.p, \boldsymbol{?SVtoPi}, \boldsymbol{!PitoSV}\rangle\} \circ h(busy, p)(s)$

(4′) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p), Suspend, \boldsymbol{?SVtoPi}, \boldsymbol{!PitoSV}\rangle\} \circ s)$
$\quad = \{PtoPZ \mapsto \langle ?PZtoP, !PtoPZ\rangle,$
$\qquad PitoQ \mapsto \langle Pi, !DtoPi\rangle\} \circ h(ready, rt.p)(s)$

Für $\#p > 0$:

(6) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p), PageFault, ?SVtoPi, !PitoSV\rangle\} \circ s)$
$\quad = \{PtoPZ \mapsto \langle ?PZtoP, !PtoPZ\rangle\} \circ h(waitMem, p)(s)$

(7) $h(waitMem, p)(\{SVtoPi \mapsto \langle MemOk\rangle\} \circ s)$
$\quad = \{PitoQ \mapsto \langle Pi, !DtoPi\rangle\} \circ h(ready, p)(s)$

5.8.2 Ein Prozessor mit Speicherzugriff

Sobald der Prozessor einem Prozeß Pi zugeteilt ist, werden die dem Prozeß zugeordneten Instruktionen in Kombination mit einem Speicherzugriff ausgeführt. Den Instruktionen sind gemäß der bereits vorgestellten Erweiterung virtuelle Adressen zugeordnet. Wir werden hier *nicht* modellieren, *wie* Berechnungen tatsächlich ausgeführt werden, sondern nur die Maßnahmen, die bzgl. der Speicherverwaltung für eine Instruktion auszuführen sind.

Daher werden die Speicherzugriffe nicht in *Lesen* und *Schreiben* aufgeteilt, sondern abstrakt als Speicheroperationen $(Step_i, ra_i)$ mit zugeordneter physikalischer Adresse $ra \in AS$ dargestellt. Um eine solche Operation ausführen zu können, benötigt der Prozessor die Informationen darüber, ob die Seite eingelagert ist, und falls ja, die zugeordnete physikalische Adresse. Bei Kopplung von Prozeß und Prozessor binden wir den lokalen Speicherverwalter an den Prozessor, so daß die relevante Information bzgl. der Realisierung einer virtuellen Adresse direkt an den Prozessor gegeben werden kann. Ein Prozessor durchläuft, bezogen auf einen Prozeß Pi, zwei Phasen, die in Abbildung 5.8.3 graphisch veranschaulicht sind.

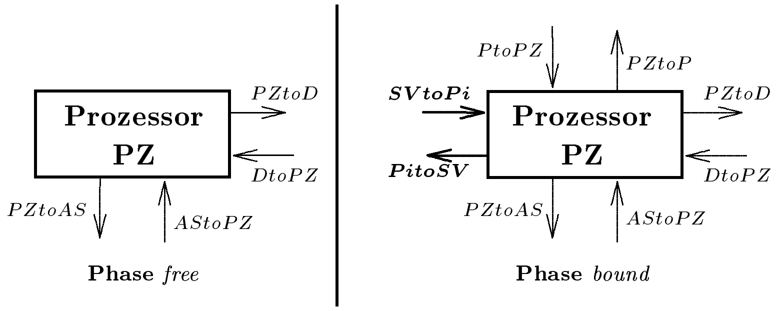


Abbildung 5.8.3: Prozessor mit Speicherzugriff

Initial ist der Prozessor, entsprechend zur oben gezeigten Phase *free*, mit keinem Prozeß verbunden. Abbildung 5.8.4 beschreibt die initiale Vernetzung von PZ.

agent PZ
  input channels $DtoPZ : S_{DtoPZ}, \boldsymbol{AStoPZ : S_{AStoPZ}}$
  output channels $PZtoD : S_{PZtoD}, \boldsymbol{PZtoAS : S_{PZtoAS}}$
  private channels $\emptyset$
is basic
  $\boldsymbol{f_{PZ}}$ mit der Spezifikation von Seite 60 mit der Erweiterung von Seite 136
end PZ

Abbildung 5.8.4: ANDL-Spezifikation von PZ mit Speicherzugriff

Die den Kanälen zugeordneten Nachrichtentypen wurden bereits in den Spezifikationen der vorangehenden Abschnitten festgelegt. Mit den Forderungen an das Verhalten des Prozessors geben wir nun, ausgehend von der in Abschnitt 4.3.4 bereits gezeigten Spezifikation eines Prozessors, folgende textuelle Beschreibung an.

(2.1) PZ erhält über Kanal $PtoPZ$ entweder nur die Nachricht $(Step_i, va_i)$ oder **zusätzlich die Ports ?*SVtoPi* und !*PitoSV***. PZ sendet die Nachricht $\boldsymbol{va_i}$ über **Kanal *PitoSV*** und geht in den Zustand $(Step_i, va_i)$ über.

(2.2) PZ erhält im Zustand $(Step_i, va_i)$ über Kanal $SVtoPi$ die Nachricht (va_i, ra_i).

(a) PZ sendet die **Nachricht** $\boldsymbol{(Step_i, ra_i)}$ über **Kanal** $\boldsymbol{PZtoAS}$ und über Kanal $PZtoP$ die Nachricht $Ok(Step_i, va_i)$.

(b) Erhält PZ zusätzlich über Kanal $DtoPZ$ die Nachricht $Suspend$, werden die **Nachricht** $\boldsymbol{(Step_i, ra_i)}$ über **Kanal** $\boldsymbol{PZtoAS}$ und die Nachrichten $Ok(Step_i, va_i)$, $Suspend$ sowie die **Ports** $\boldsymbol{?SVtoPi}$ und $\boldsymbol{!PitoSV}$ über Kanal $PZtoP$ gesendet.

(2.3) Erhält PZ im Zustand $(Step_i, va_i)$ über Kanal $DtoPZ$ die Nachricht $Suspend$, so geht er in den Zustand $(Suspend, Step_i(va_i))$ über.

(3.1) PZ erhält über Kanal $PtoPZ$ entweder nur die Nachricht $(Step_i, va_i)$ oder zusätzlich die **Ports** $\boldsymbol{?SVtoPi}$ und $\boldsymbol{!PitoSV}$ sowie über Kanal $DtoPZ$ die Nachricht $Suspend$. PZ sendet die **Nachricht** $\boldsymbol{va_i}$ über **Kanal** $\boldsymbol{PitoSV}$ und geht in den Zustand $(Suspend, Step_i(va_i))$ über.

(3.2) Erhält PZ im Zustand $(Suspend, Step_i(va_i))$ über Kanal $SVtoPi$ die Nachricht (va_i, ra_i), werden die **Nachricht** $\boldsymbol{(Step_i, ra_i)}$ über **Kanal** $\boldsymbol{PZtoAS}$ und $Ok(Step_i, va_i)$ sowie die **Ports** $\boldsymbol{?SVtoPi}$ und $\boldsymbol{!PitoSV}$ über Kanal $PZtoP$ gesendet.

(7) Erhält PZ in einem der Zustände $(Step_i, va_i)$ oder $(Suspend, (Step_i, va_i)$ über Kanal $SVtoPi$ die Nachricht $PageFault$, werden die Nachricht $PageFault$ und die Ports $?SVtoPi$ und $!PitoSV$ über Kanal $PZtoP$ gesendet.

Die Numerierung der einzelnen Anforderungen orientiert sich an der von Abschnitt 4.3.4. Die Anpassungen der Spezifikation der Komponente PZ betreffen erwartungsgemäß die Ausführung eines Berechnungsschrittes. Daher sind i.w. die in der Spezifikation aus Abschnitt 4.3.4 mit (2) und (3) bezeichneten Schritte zu modifizieren. Insgesamt muß die Erweiterung der Instruktionen um eine virtuelle Adresse berücksichtigt werden. Sobald Prozeß und Prozessor gekoppelt sind, gibt Pi seinen lokalen Speicherverwalter an der Prozessor weiter und mit der Suspendierung wird diese Bindung gelöst.

Zur Anpassung an die Spezifikation aus Abschnitt 4.3.4 haben wir die Erweiterungen hier mit (2.1) bis (2.3) sowie (3.1) und (3.2) bezeichnet. Die Ausführung einer Instruktion umfaßt nun nicht mehr nur die Meldung $Ok(Step_i)$, sondern die Überprüfung und Umrechnung einer virtuellen Adresse, die Ausführung der Instruktion auf dem Arbeitsspeicher und erst abschließend die Weitergabe von $Ok(Step_i)$ an den Prozeß. Während der Durchführung eines Schrittes und den dabei erforderlichen Interaktionen ist generell der Tatsache Rechnung zu tragen, daß der Prozessor jederzeit suspendiert werden kann und dies auch durchzuführen ist.

Der mit (7) bezeichnete Schritt beschreibt das für das Verhalten vollständig neue Auftreten eines Seitenfehlers. Aufgrund eines Seitenfehlers werden der Prozessor freigegeben und der Prozeß blockiert. Basierend auf der bereits erstellten Spezifikation des lokalen Speicherverwalters in Abschnitt 5.5 ist bekannt, daß die Speicherverwalter dafür sorgen, daß die

Ursache für einen Seitenfehler behoben wird. In unserem System ist der Prozessor hiervon jedoch nicht betroffen.

Die wesentlichen Anpassungen und Veränderungen relativ zur Spezifikation des Prozessors in Abschnitt 4.3.4 werden in der folgenden Umsetzung **markiert**.

Funktionsgleichungen für f_{PZ}

$$\forall s \in \prod_{n \in N_{Mem}} [S_n^*], \quad Step_i \in STEP, \quad \boldsymbol{va_i} \in \bigcup_{\boldsymbol{j=1}}^{\boldsymbol{n}} \boldsymbol{VA_{Pj}}, \quad \boldsymbol{ra_i} \in \boldsymbol{ASA},$$
$$m \in \{\langle (Step_i, va_i)\rangle, \ \langle (Step_i, va_i), \boldsymbol{?SVtoPi, !PitoSV} \rangle\} \ :$$
$$\exists\ g \in (STEP \times VA_{Pi}) \ \rightarrow \ Type_{PZ}\ , \quad h \in \big(\{Suspend\} \ \times \ (STEP \times VA_{Pi})\big) \ \rightarrow \ Type_{PZ} \ :$$

(2.1) $f_{PZ}(\{PtoPZ \mapsto m, DtoPZ \mapsto \langle\rangle\} \circ s)$
$= \{\boldsymbol{PitoSV} \mapsto \langle \boldsymbol{va_i} \rangle\} \circ g(Step_i, va_i)(s)$

(2.2a) $g(Step_i, va_i))(\{\boldsymbol{SVtoPi} \mapsto \langle \boldsymbol{(va_i, ra_i)} \rangle, DtoPZ \mapsto \langle v \rangle\} \circ s)$
$= \{\boldsymbol{PZtoAS} \mapsto \langle \boldsymbol{(Step_i, ra_i)} \rangle, PZtoP \mapsto \langle Ok(Step_i, va_i) \rangle\} \circ f_{PZ}(s)$

(2.2b) $g(Step_i, va_i))(\{\boldsymbol{SVtoPi} \mapsto \langle \boldsymbol{(va_i, ra_i)} \rangle, DtoPZ \mapsto \langle Suspend \rangle\} \circ s)$
$= \{\boldsymbol{PZtoAS} \mapsto \langle \boldsymbol{(Step_i, ra_i)} \rangle,$
$PZtoP \mapsto \langle Ok(Step_i, va_i), Suspend, \boldsymbol{?SVtoPi, !PitoSV} \rangle\} \circ f_{PZ}(s)$

(2.3) $g(Step_i, va_i))(\{SVtoPi \mapsto \langle\rangle, DtoPZ \mapsto \langle Suspend \rangle\} \circ s) \ = \ h(Suspend, (Step_i, va_i))(s)$

(3.1) $f_{PZ}(\{PtoPZ \mapsto m, DtoPZ \mapsto \langle Suspend \rangle\} \circ s)$
$= \{\boldsymbol{PitoSV} \mapsto \langle \boldsymbol{va_i} \rangle\} \circ h(Suspend, (Step_i, va_i))(s)$

(3.2) $h(Suspend, (Step_i, va_i))(\{\boldsymbol{SVtoPi} \mapsto \langle \boldsymbol{(va_i, ra_i)} \rangle\} \circ s)$
$= \{\boldsymbol{PZtoAS} \mapsto \langle \boldsymbol{(Step_i, ra_i)} \rangle,$
$PZtoP \mapsto \langle Ok(Step_i, va_i), Suspend, \boldsymbol{?SVtoPi, !PitoSV} \rangle\} \circ f_{PZ}(s)$

(7a) $g(Step_i, va_i)(\{SVtoPi \mapsto \langle PageFault \rangle\} \circ s)$
$= \{PZtoP \mapsto \langle PageFault, ?SVtoPi, !PitoSV \rangle\} \circ f_{PZ}(s)$

(7b) $h(Suspend, (Step_i, va_i))(\{SVtoPi \mapsto \langle PageFault \rangle\} \circ s)$
$= \{PZtoP \mapsto \langle PageFault, ?SVtoPi, !PitoSV \rangle\} \circ f_{PZ}(s)$

Kapitel 6

Prozeßkooperation

Mit den bisher entwickelten System sind zwei wesentliche Teilbereiche eines Betriebssystems auf abstrakte Weise modelliert: die Prozessor- und die Speicherverwaltung. Die Ausführung einer Berechnung schreitet sukzessive voran und wird unterbrochen, wenn der Prozessor und/oder der benötigte Speicherplatz nicht verfügbar ist. Es wird dafür gesorgt, daß die Ursache für die Unterbrechung behoben wird und die Ausführung der Berechnung fortgesetzt werden kann. Mit dem nächsten Entwicklungsschritt können die bisher voneinander unabhängigen Prozesse kooperieren und müssen sich dafür möglicherweise untereinander abstimmen. Das Verhalten unseres Systems wird um die Konzepte des Nachrichtenaustauschs mit *blockierendem Senden* und alternativ dazu *nichtblockierendem Senden* erweitert. Wir werden zunächst die Konzepte zur Prozeßkooperation und insbesondere des Nachrichtenaustauschs vorstellen, die gewählte Vorgehensweise im Groben erläutern und anschließend die Erweiterung der Systemspezifikation im einzelnen durchführen.

6.1 Einführung

Im Rahmen der Prozessor- und Speicherverwaltung wird dafür gesorgt, daß die Berechnungen aller Prozesse fortschreiten. Dies konnte auch an voneinander unabhängigen Prozessen demonstriert werden. Da Prozesse jedoch Problemlösungen auch in Kooperation erarbeiten sollten, werden Abstimmungen der Prozesse untereinander notwendig. Eine einfache Lösung hierfür besteht darin, daß ein Prozeß das Ergebnis verwendet, das ein anderer Prozeß bei seiner Terminierung liefert, vergleiche beispielsweise Kapitel 9.2 in [Fin88]. In Betriebssystemen werden zwei Arten der Prozeßkooperation alternativ realisiert:

- Kooperation durch *gemeinsamen Speicher* und
- Kooperation durch *Nachrichtenaustausch*.

6.1.1 Gemeinsamer Speicher

Die Verwendung von gemeinsamem Speicher bedeutet, daß sich Prozesse bestimmte Variablen bzw. Speicherbereiche teilen und diese gemeinsam benutzen. Hierbei werden die Abläufe, in denen die Nutzung der gemeinsamen Daten von der zeitlichen Reihenfolge der Nutzung durch die Prozesse abhängig ist, als *zeitkritische Abläufe* bezeichnet, siehe die Abschnitte 2.2.1 in [Tan94], 9.5 in [PS85] und Kapitel 2 in [MOO87]. Bei der Nutzung *gemeinsamer Daten* ergibt sich das Problem, daß mehrere Prozesse möglicherweise zum gleichen Zeitpunkt auf einen gemeinsamen Speicherbereich zugreifen. Dabei können Konfliktsituationen auftreten, in denen ein Prozeß den Wert der Daten verändert, während ein anderer zum gleichen Zeitpunkt den Wert der Daten liest. Um dies auszuschließen, muß der sogenannte *wechselseitige Ausschluß* gewährleistet sein. Greift ein Prozeß auf einen gemeinsamen Speicherbereich zu, wird auf diese Weise sichergestellt, daß alle anderen Prozesse den Speicherbereich zu diesem Zeitpunkt auf keinen Fall nutzen. Die Auswahl der geeigneten Realisierung ist eine wichtige Entwurfsentscheidung in Betriebssystemen.

Die verschiedenen Lösungswege werden entsprechend zu den bereits beschriebenen Strategien des Scheduling oder der Seitenersetzungen gemäß spezieller Kriterien bewertet. Die Lösungen zur Gewährleistung von wechselseitigem Ausschluß reichen über *aktives Warten* (Abschnitt 2.2.3 in [Tan94]), *Semaphore* (Abschnitte 2.2.5 in [Tan94] und 9.6 in [PS85]) bis hin zu *Monitoren* (Abschnitte 2.2.7 in [Tan94] und 2.9 in [Fin88]). Da wir in der vorliegenden Arbeit weder Daten noch Algorithmen auf Datenbereichen modellieren, gehen wir auf diese Konzepte und Lösungen nicht weiter ein. Die genannte Literatur behandelt diese Probleme mit großer Ausführlichkeit. Da Focus vor allem das Konzept des Nachrichtenaustausch verwendet, werden wir das im folgenden Abschnitt beschriebene Konzept der nachrichtenorientierten System modellieren.

6.1.2 Nachrichtenaustausch

Neben der Benutzung von gemeinsamem Speicher können Prozesse durch direkten Nachrichtenaustausch (*Message Passing*) miteinander kooperieren. Die Prozesse stehen in *Sender*- und *Empfänger*-Beziehung zueinander. Die Konzepte des Nachrichtenaustauschs werden beispielsweise in Abschnitt 2.2.8 von [Tan94] sowie die Kapitel 2 in [Spi97b] und Abschnitt 9.8 in [PS85] beschrieben.

Die Konzepte des Nachrichtenaustauschs werden darin unterschieden, wie Sender und Empfänger einander zugeordnet sind: Mit der 1:1-Zuordnung sind jeweils der Sender und der Empfänger eindeutig festgelegt. Daneben gibt es m:1-Zuordnungen, in denen mehrere Sender und 1:n-Zuordnungen, wobei mehrere Empfänger vorliegen, sowie die allgemeine Form der m:n-Zuordnung, in denen beliebige Zuordnungen getroffen werden können. Insbesondere muß festgelegt werden, ob jeder im System auftretende Prozeß mit jedem anderen Prozeß kommunizieren kann, oder ob hier einschränkende Festlegungen getroffen werden.

Weitere Varianten sind dadurch gekennzeichnet, wie die Kommunikation die Ausführung der Berechnung beeinflußt. Es wird unterschieden zwischen:

- Die Kommunikation erfolgt in Abstimmung; erst wenn Sender und Empfänger bereit zur Kommunikation sind, kann die stattfinden. Diese Form des Nachrichtenaustauschs wird als *blockierendes Senden* (mittels Rendezvous) bezeichnet.
- Die Kommunikation kann erfolgen, ohne daß sich Sender und Empfänger abstimmen. Diese Form wird *nichtblockierendes Senden* genannt.

Blockierendes und nichtblockierendes Senden sind alternative Primitive, wobei die Entscheidung für eine der Varianten vom Systementwickler getroffen wird. Nur selten werden beide Arten angeboten, so daß der Benutzer zwischen ihnen wählen kann.

Als Basisoperationen werden eine Sende- und eine Empfangsoperation für alle Prozesse einheitlich festgelegt. Wir erklären sie hier für die Prozesse Pi und Pj mit $i, j \in \{1, \ldots, n\}$ und $i \neq j$ sowie Nachrichten des allgemein festgelegten Typs Msg.

Sendeanweisung : Pi : `send` *exp* `to` Pj;

(*exp* ist ein Ausdruck vom Typ *Msg*)

Empfangsanweisung : Pj : `receive` *var* `from` Pi;

(*var* ist eine Variable vom Typ *Msg*)

Die 1:1-Zuordnung kann anhand der Anweisungen nachvollzogen werden: Mit der Sendeanweisung wird festgelegt, daß der Wert des Ausdrucks *exp* von Prozeß Pi an den Prozeß Pj („`to` Pj") zu senden ist. Mit der Empfangsoperation wird gekennzeichnet, daß der Prozeß Pj eine Nachricht von Prozeß Pi („`from` Pi") empfangen soll. Mit diesen einheitlichen Festlegungen für die Sende- und Empfangsoperationen beschreiben wir die oben genannten Varianten *blockierendes* und *nichtblockierendes Senden.*

6.1.2.1 Blockierendes Senden (Rendezvous)

Wir beschreiben *blockierendes Senden* für die beiden Prozesse Pi und Pj mittels eines Rendezvous. Die Sendeoperation sei für Prozeß Pi und die entsprechende Empfangsoperation für Prozeß Pj definiert. Es gilt:

1. Von Pi wird die Sendeoperation ausgeführt; Pi bestimmt den Wert von *exp*, sendet diesen an Pj und wartet auf die Empfangsbestätigung von Pj.
2. Von Pj wird die Empfangsoperation ausgeführt. Pj überprüft das Vorliegen einer Nachricht von Pi. Falls diese noch nicht vorliegt, wird Pj bis zum Eintreffen der Nachricht blockiert.
3. Das *Rendezvous* wird ausgeführt, sobald die Nachricht bei Pj vorliegt. Pj übernimmt den Wert der Nachricht in *var* und bestätigt Pi deren Empfang. Damit sind die

Sende- bzw. Empfangsoperationen für beide Prozesse ausgeführt, und sie können ihre Berechnungen unabhängig voneinander fortsetzen.

Sender und Empfänger schließen den Nachrichtenaustausch im Rendezvous gemeinsam ab. Der Empfänger Pj hat die Nachricht empfangen, und der Sender Pi weiß, daß Pj die Nachricht empfangen hat.

6.1.2.2 Nichtblockierendes Senden

Die zweite alternative Vorgehensweise für Nachrichtenaustausch besteht im nichtblockierenden Senden, bei dem sich die beiden Partner nicht abstimmen müssen. Für Pi sei die Sende- und für Pj die Empfangsoperation definiert. Das nichtblockierende Senden und Empfangen wird auf folgende Weise ausgeführt:

1. Pi führt die Sendeoperation aus, bestimmt den Wert von *exp* und sendet diesen an Pj. Damit ist die Ausführung der Sendeanweisung für Pi abgeschlossen.

2. Pj führt eine Empfangsoperation aus und überprüft das Vorliegen einer Nachricht von Pi. Liegt diese nicht vor, wird Pj bis zum Eintreffen der Nachricht blockiert. Sobald die Nachricht vorliegt, übernimmt Pj diese in *var* und wird entblockiert.

In diesem Fall kann der Sender unabhängig vom Empfänger die Ausführung seiner Berechnung fortsetzen, wobei der Sender konzeptionell keine Kenntnis davon erhält, ob der Empfänger die Nachricht erhalten hat, oder nicht.

6.2 Methodische Vorgehensweise

Bei der Beschreibung der Anforderungen an das System und Konzepte, zu denen wir in den Kapiteln 4 und 5 die formalen Modellierungen entwickelt haben, sind wir davon ausgegangen, daß die im System vorhandenen Prozesse unabhängig voneinander arbeiten und für die Durchführung ihrer Berechnungen ausschließlich einen Prozessor und in der ersten Erweiterung zusätzlich Speicher benötigen. Das bekannte Zustandsdiagramm für Prozesse, siehe Abbildung 3.2.1 in Kapitel 3, wurde daher zunächst auf die beiden Zustände *ready* und *busy* reduziert. In Kapitel 5 wurde ein Wartezustand *waitMem* hinzugenommen, siehe Abbildung 5.8.1. Der im allgemeinen Zustandsdiagramm aufgeführte Zustand *waiting* umfaßt alle Wartezustände, in denen ein Prozeß auf ein, vom Prozessor verschiedenes, Betriebsmittel wartet. Dieses Betriebsmittel wird von ihm zur Weiterführung der Berechnung benötigt und steht für ihn aktuell nicht zur Verfügung. Zu den Ursachen der Blockierung eines Prozesses gehört insbesondere auch, daß er auf eine Nachricht wartet, wenn die Prozesse kooperieren. Wir nehmen zu dem bisher gültigen Zustandsdiagramm von Abbildung 5.8.1 den Zustand *waitKoop* hinzu und erhalten das in Abbildung 6.2.1 gezeigte Zustandsdiagramm für Prozesse.

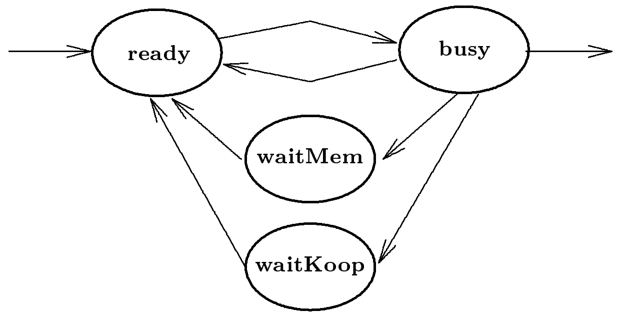


Abbildung 6.2.1: Zustandsdiagramm für kooperierende Prozesse

Mit den im folgenden gezeigten Erweiterungen kann ein Prozeß mit anderen Prozessen kooperieren. Hierbei wählen wir die in Abschnitt 6.1.2 erklärte Kooperationsform mittels Nachrichtenaustausch. Die Prozesse werden zu Gruppen, wir nennen sie *Familien*, zusammengeschlossen, wobei jeder Prozeß mit jedem Mitglied dieser Familie mittels Nachrichtenaustausch kooperieren kann. Ausgehend von dem in Kapitel 4 entwickelten System mit einem Prozessor und n Prozessen und dessen Erweiterungen in Kapitel 5 erhalten wir die folgende Aufgabenstellung:

> Für ein Einprozessorsystem wird neben der Verwaltung der Betriebsmittel „Prozessor“ und „Speicher“ die Prozeßkooperation mittels Nachrichtenaustausch und den beiden Alternativen „blockierendes“ und „nichtblockierendes Senden“ modelliert.

Das System, auf dem unsere Erweiterung basiert, besteht aus n Prozessen, dem Prozessor, Arbeits- und Hintergrundspeicher, den Komponenten der Prozessorverwaltung (Dispatcher, Timer und Queue) sowie den Komponenten der Speicherverwaltung (globaler und lokale Speicherverwalter). Zur Behandlung der oben beschriebenen Aufgabe erweitern wir das System, analog zu unserer bisherigen Vorgehensweise, um *Kooperationsverwalter*. Diese sind jeweils einer Prozeßfamilie zugeordnet und koordinieren den Nachrichtenaustausch zwischen den Prozessen einer Familie. Insgesamt erhalten wir mit den in diesem Kapitel vorgenommen Erweiterungen das in Abbildung 6.2.2 im Groben dargestellte verteilte System.

Für die Prozesse und den Prozessor ist zu beachten:

Prozesse: Der Nachrichtenaustausch findet mittels der bereits erklärten Sende- und Empfangsoperationen statt. Bisher haben wir die Beschreibung der Berechnung, die durch einen Prozeß ausgeführt wird, nicht detaillierter betrachtet. Sie wurde abstrakt als Folge von Berechnungsschritten $Step_i$ mit jeweils zugeordneter virtueller Adresse va_i modelliert. Durch die Hinzunahme der speziellen Kooperationsanweisungen wird die Beschreibung des Programms erweitert.

Den Nachrichtenaustausch mittels *Rendezvous* machen wir unter Verwendung der Konzepte des mobilen FOCUS explizit sichtbar, indem zwischen sendendem und empfangenden Prozeß Kooperationskanäle erzeugt werden, über die das Rendezvous direkt ausgeführt wird.

Prozessor: Entsprechend zu den Erweiterungen der Beschreibung eines auszuführenden Programms werden die Prozessoren angepaßt. Insbesondere entspricht die Ausführung der Sende- und Empfangsoperationen der Ausführung von Lese- und Schreiboperationen auf dem Arbeitsspeicher. Analog zu den bisher gezeigten Modellierungen wird der Kooperationsverwalter eines Prozesses bei Zuteilung des Prozessors an diesen weitergegeben.

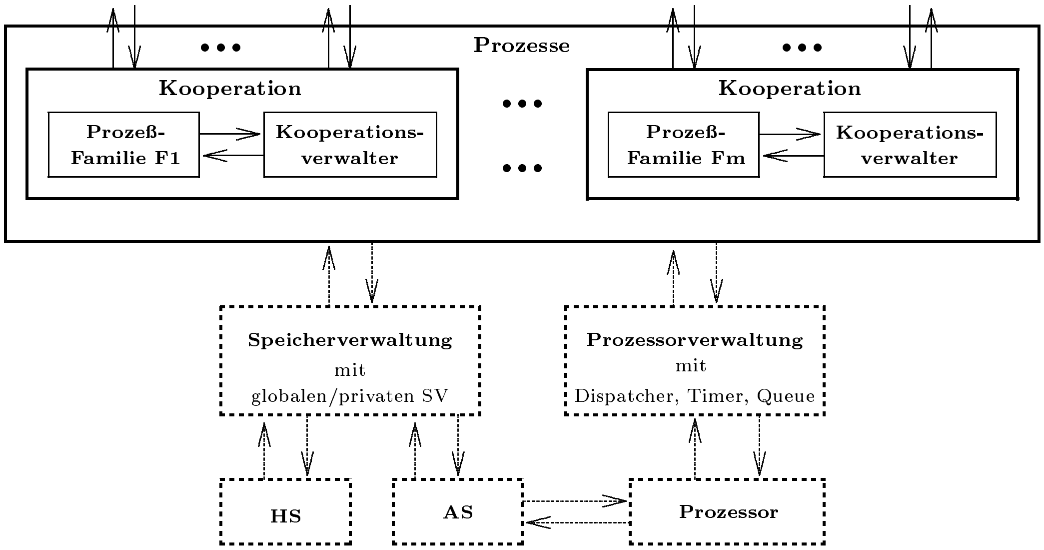


Abbildung 6.2.2: Erweiterung des Systems um Prozeßkooperation

Im folgenden werden das bisher modellierte System um *Kooperationsverwalter* erweitert und die Spezifikationen des Prozessors und der Prozesse gemäß den beschriebenen Anforderungen ergänzt. Entsprechend dem Einsatz von blockierendem und nichtblockierendem Senden in Betriebssystemen erarbeiten wir zwei alternative Modellierungen.

6.3 Ein System mit kooperierenden Prozessen

Das System mit n Prozessen, den Komponenten zur Prozessorverwaltung und zur Speicherverwaltung wird um Komponenten erweitert, die die Kommunikationen zwischen den Prozessen koordinieren. Prozesse werden zu Familien zusammengeschlossen, wobei ein Prozeß ausschließlich mit den Mitgliedern *seiner* Familie kooperieren kann. Wir gehen davon aus, daß jede Familie eine feste Anzahl von f Prozessen umfaßt. Zur Vereinfachung der Notationen in den SSDs, der Kanalbezeichner und somit in ANDL-Spezifikation setzen wir weiterhin voraus, daß die Anzahl n der Prozesse von Beginn an festgelegt und ein Vielfaches

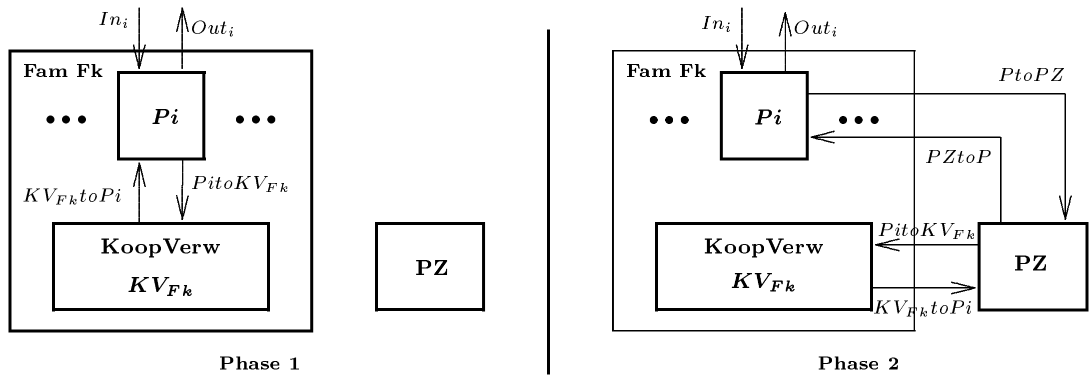


Abbildung 6.3.3: Das System `Koop` mit kooperationsfähigen Prozessen

von f ist. In unserer Modellierung seien m Familien enthalten; es gilt $n = m * f$. Abbildung 6.3.3 zeigt beispielhaft anhand der Familie Fk die Kopplung der Prozesse mit *ihrem* Kooperationsverwalter. Im SSD zeigen wir die *Phase 1*, in der Pi blockiert ist oder auf die Zuteilung des Prozessors wartet, und die *Phase 2*, in der dem Prozeß Pi der Prozessor zugeteilt ist. Bereits in Abschnitt 6.2 haben wir erklärt, daß Pi *seinen* Kooperationsverwalter an den Prozessor weitergibt, sobald ihm dieser zugeteilt wird.

agent Koop
  input channels $\ldots, In_i : S_{In_i}, \ldots$
  output channels $\ldots, Out_i : S_{Out_i}, \ldots$
is network

$$
\begin{array}{lcl}
\ll Out_1, \boldsymbol{P1toKV_{F1}}, \ldots \gg & = & \mathtt{P1} \ll In_1, \boldsymbol{KV_{F1}toP1}, \ldots \gg \;; \\
\vdots & \vdots & \vdots \\
\ll Out_n, \boldsymbol{PntoKV_{Fm}}, \ldots \gg & = & \mathtt{Pn} \ll In_n, \boldsymbol{KV_{Fm}toPn}, \ldots \gg \;; \\
\ll KV_{F1}toP1, \ldots, KV_{F1}toPf \gg & = & \mathtt{KV}_{F1} \ll P1toKV_{F1}, \ldots, PftoKV_{F1} \gg \;; \\
\vdots & \vdots & \vdots \\
\ll KV_{Fm}toP1_m, \ldots, KV_{Fm}toPn \gg & = & \mathtt{KV}_{Fm} \ll P1_m toKV_{Fm}, \ldots, PntoKV_{Fm} \gg \;;
\end{array}
$$

$$[sowie\ die\ Komponenten: Prozessor,\ Dispatcher,\ Queue\ und\ Timer]$$

$$[private/globale\ Speicherverwalter,\ Arbeits-\ und\ Hintergrundspeicher]$$

end Koop

Abbildung 6.3.4: ANDL-Spezifikation des Systems mit kooperierenden Prozessen

In Abbildung 6.3.3 haben wir darauf verzichtet, die Komponenten darzustellen, die von der Erweiterung der Funktionalitäten des Systems nicht direkt betroffen sind, wie die Speicherverwaltung, Dispatcher, Queue und Timer. Die Einbettung des hier gezeigten SSD in den bisherigen Systemaufbau ergibt sich mit Abbildung 6.2.2. Abbildung 6.3.4 zeigt die ANDL-Spezifikation des Systems `Koop`. Auch hier verzichten wir auf die Kanäle und

Komponenten, die aus den Kapiteln 4 und 5 bekannt sind, und die für die Erweiterung des Systems um die Kooperationsfähigkeit der Prozesse nicht relevant sind bzw. deren Spezifikation unverändert übernommen werden kann. Wir gehen in der ANDL-Spezifikation davon aus, daß Prozeß $P1$ der Familie $F1$ und Prozeß Pn der Familie Fm zugeordnet ist. Die Menge der Kanalbezeichner wird festgelegt durch die Menge N_{Koop}.

6.4 Blockierendes Senden

Zur Modellierung des in Abschnitt 6.1.2 erläuterten Konzepts *blockierendes Senden mit Rendezvous* werden wir die Möglichkeiten mobiler Systeme ausnutzen. Die Prozesse, die miteinander kooperieren, müssen sich beim Senden bzw. Empfangen einer Nachricht untereinander abstimmen. Damit wird sichergestellt, daß die gesendete Nachricht tatsächlich empfangen wurde. Um diese Abstimmung zu gewährleisten, modellieren wir den Aufbau einer eigens hierfür vorgesehenen Kommunikationsverbindung zwischen dem sendenden und dem empfangenden Prozeß, die nach erfolgreichem Abschluß der Kommunikation wieder gelöscht wird. Für die Koordination des Aufbaus dieser Verbindung ist der Kooperationsverwalter zuständig. Das *Rendezvous* führen die Prozesse unabhängig vom Verwalter aus. Wir nennen einen Prozeß, der eine Sendeoperation ausführt, *Sender*, und einen Prozeß, der eine Empfangsoperation ausführt, *Empfänger*.

Prozesse können ausschließlich mit solchen Prozessen kooperieren, die Mitglieder ihrer Familie sind. Zu jeder Familie gehört ein Kooperationsverwalter, der die Kooperationen zwischen den Prozessen koordiniert. Ein Kooperationsverwalter könnte so modelliert werden, daß er Blockierungen (*Deadlocks*) vermeidet, wenn ein Prozeß mit einem anderen, nicht zur Familie gehörenden, Prozeß kooperieren möchte. Wir gehen für unsere Modellierung jedoch davon aus, daß dieser Fall nicht eintritt und werden die Aufgabe der Deadlockvermeidung nicht betrachten. Die Menge der Prozesse und die Aufteilung in Familien seien für das System initial festgelegt. In den folgenden Abschnitten entwickeln wir die formale Modellierung des Kooperationsmanagers sowie der erweiterten, kooperationsfähigen Prozesse und geben zum Abschluß die Erweiterung des Prozessors an.

6.4.1 Der Kooperationsverwalter für Rendezvous

Wir modellieren repräsentativ einen Kooperationsverwalter KV_{F1}, der für die Koordinierung des Nachrichtenaustauschs der Familie $F1$ zuständig ist, und gehen davon aus, daß die Prozesse $P1$ bis Pf zu dieser Familie gehören. Der Kooperationsverwalter koordiniert den Nachrichtenaustausch zwischen diesen Prozessen. Nachrichten, die von einem Sender verschickt werden, werden geeignet zwischengespeichert und können vom Empfänger abgefragt werden. Ein Prozeß, der eine Empfangsoperation ausführt, und für den die gewünschte Nachricht noch nicht vorliegt, wird blockiert, und es wird registriert, daß er empfangsbereit ist. Ein Verwalter ist für folgende Aufgaben zuständig:

- Die korrekte Zuordnung der Nachrichten und Empfangsbereitschaften zu den jeweiligen Partnern.
- Die Speicherung gesendeter Nachrichten, deren Weitergabe an einen empfangsbereiten Empfänger sowie die Registrierung der Empfangsbereitschaft eines Prozesses.

Diese Anforderungen werden für eine Menge von f Prozessen gewährleistet, wobei ein Prozeß Pi mit jedem anderen Prozeß Pj mit $j \in \{1, \ldots, f\} \setminus \{i\}$ Nachrichten austauschen kann. Entsprechend zu bereits durchgeführten Modellierungen, wie beispielsweise die des globalen Speicherverwalters in Abschnitt 5.6, wird ein Kooperationsverwalter als verteiltes System, siehe Abbildung 6.4.1 modelliert, dessen Subkomponenten für die oben beschriebenen Aufgaben zuständig sind.

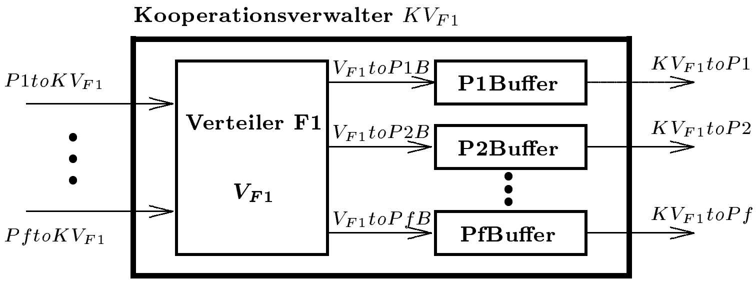


Abbildung 6.4.1: Ein Kooperationsmanager für Rendezvous

Das in Abbildung 6.4.1 für die Komponente KV_{F1} gezeigte SSD wird in die in Abbildung 6.4.2 vorgestellte ANDL-Spezifikation umgesetzt. Die für die Kanäle gültigen Nachrichtentypen werden in den folgenden Abschnitten festgelegt.

agent $\mathtt{KV}_{F1}$

input channels $P1toKV_{F1} : S_{P1toKV_{F1}}, \ldots, PftoKV_{F1} : S_{PftoKV_{F1}}$

output channels $KV_{F1}toP1 : S_{KV_{F1}toP1}, \ldots, KV_{F1}toPf : S_{KV_{F1}toPf}$

is network

$$\begin{array}{lcl}
\ll V_{F1}toP1B, \ldots, V_{F1}toPf \gg & = & \mathtt{V}_{F1} \ll KV_{F1}toP1, \ldots, KV_{F1}toPf \gg \;; \\
\ll KV_{F1}toP1 \gg & = & \mathtt{P1B} \ll V_{F1}toP1B \gg \;; \\
\vdots & \vdots & \quad \vdots \\
\ll KV_{F1}toPf \gg & = & \mathtt{PfB} \ll V_{F1}toPfB \gg \;;
\end{array}$$

end $\mathtt{KV}_{F1}$

Abbildung 6.4.2: ANDL-Spezifikation des Kooperationsverwalters $\mathtt{KV}_{F1}$

6.4.1.1 Die Verteilerkomponente

Die Komponente V_{F1} ist dafür zuständig, die Aufforderungen zum Nachrichtenaustausch aller Prozesse der Familie entgegenzunehmen. Hierzu gehören sowohl gesendete Nachrichten als auch die Bereitschaft, eine Nachricht vom Kommunikationspartner zu empfangen. V_{F1} ist mit den Prozessen $P1$ bis Pf jeweils durch einen Kanal $PitoKV_{F1}$ mit $i \in \{1, \ldots, f\}$, auf den sie lesend zugreifen kann, verbunden. Über $PitoKV_{F1}$ empfängt V_{F1} die Nachrichten $Send(val, Pj)$ bzw. $Rec(Pj)$, die wir abkürzend für die in Abschnitt 6.1.2 definierten Basisoperationen verwenden. Beide Nachrichten enthalten jeweils einen Identifikator Pj für den Partner des Nachrichtenaustauschs, wobei wir voraussetzen, daß Pj mit $j \in \{1, \ldots, f\}$ zur Familie $F1$ gehört. Der Kooperationsverwalter KV_{F1} verfügt für jeden Prozeß Pi über einen Speicher, in dem zu diesem gesendete Nachrichten und die Empfangsbereitschaft registriert werden. V_{F1} ordnet die Nachrichten, die sie von den Prozessen erhält, in den passenden Speicher (Puffer) ein. Ausgehend von dem in Abbildung 6.4.1 gezeigten SSD für KV_{F1} erhalten wir die in Abbildung 6.4.3 gezeigte ANDL-Spezifikation für V_{F1}.

agent V_{F1}
  input channels $P1toKV_{F1} \ : \ S_{P1toKV_{F1}}, \ldots, PftoKV_{F1} \ : \ S_{PftoKV_{F1}}$
  output channels $V_{F1}toP1B \ : \ S_{V_{F1}toP1B}, \ldots, V_{F1}toPfB \ : \ S_{V_{F1}toPfB}$
  private channels $\emptyset$
is basic
  $\boldsymbol{f_{V_{F1}}}$ $\boxed{\text{mit der Spezifikation von Seite 147}}$
end V_{F1}

Abbildung 6.4.3: ANDL-Spezifikation von V_{F1}

Für die verwendeten Kanälen gelten die in Tabelle 6.4.1 definierten Nachrichtentypen.

Kanal n	Nachrichtenmengen S_n
$PitoKV_{F1}$	$\left.\begin{array}{l}\{Send(val,pid) \mid val \in Msg, pid \in \{P1,\ldots,Pf\} \setminus \{Pi\}\} \\ \quad \{Rec(pid) \mid pid \in \{P1,\ldots,Pf\} \setminus \{Pi\} \\ \{Send(val,pid) \mid val \in Msg, pid \in \{P1,\ldots,Pf\} \setminus \{Pi\}\} \\ \quad \{Rec(pid) \mid Pid \in \{P1,\ldots,Pf\} \setminus \{Pi\}\end{array}\right\} \cup\ ?!N_{KoopRV}$
$V_{F1}toPiB$	

Tabelle 6.4.1: Nachrichtentypen für V_{F1}

Für das Verhalten von V_{F1} gilt:

(1) Erhält V_{F1} über Kanal $PitoKV_{F1}$ die Nachricht $Send(val, Pj)$ sowie einen Port $!ConftoPi$, wird $\langle (val, Pi), !ConftoPi \rangle$ über Kanal $V_{F1}toPjB$ gesendet.

Eine Sendeoperation von Prozeß Pi zu Prozeß Pj wird registriert, die Nachricht sowie der zugehörige Port für die Ausführung des Rendezvous werden an den Puffer für Prozeß Pj weitergeleitet.

(2) Erhält V_{F1} über Kanal $PitoKV_{F1}$ die Nachricht $Rec(Pj)$, wird die Nachricht $Rec(Pj)$ über Kanal $V_{F1}toPiB$ gesendet.

Prozeß Pi ist empfangsbereit für eine Nachricht von Prozeß Pj.

Diese Beschreibung wird in folgende formale Spezifikation umgesetzt:

Funktionsgleichungen für $f_{V_{F1}}$

$\forall s \in \prod_{n \in N_{KoopRV}} [S_n^*]$, $val \in Msg,\ Pj \in \{P1, \ldots, Pf\} \setminus \{Pi\}$:

(1) $f_{V_{F1}}(\{P1toKV_{F1} \mapsto \langle\rangle, \ldots, PitoKV_{F1} \mapsto \langle Send(val, Pj), !ConftoPi \rangle, \ldots, PftoKV_{F1} \mapsto \langle\rangle\} \circ s)$
$= \{V_{F1}toPjB \mapsto \langle (val, Pi), !ConftoPi \rangle\} \circ f_{V_{F1}}(s)$

(2) $f_{V_{F1}}(\{P1toKV_{F1} \mapsto \langle\rangle, \ldots, PitoKV_{F1} \mapsto \langle Rec(Pj) \rangle, \ldots PntoKV_{F1} \mapsto \langle\rangle\} \circ s)$
$= \{V_{F1}toPiB \mapsto \langle Rec(Pj) \rangle\} \circ f_{V_{F1}}(s)$

In beiden Gleichungen betrachten wir ausschließlich den Fall, daß nur über *einen* Kanal $PitoKV_{F1}$ eine Nachricht eintrifft, während über alle weiteren Kanäle die leere Sequenz empfangen wird. Dies ist nur im Einprozessorsystem gültig. Bei einer Erweiterung des Kooperationsverwalters für Mehrprozessorsysteme können über mehrere Kanäle Nachrichten eintreffen, die alle behandelt werden müssen. Die Modellierung könnte entsprechend zu bereits durchgeführten Spezifikationen mit einer speziellen Queue erfolgen.

6.4.1.2 Die Nachrichtenpuffer für blockierendes Senden

Zu einem Kooperationsverwalter gehören Nachrichtenpuffer, wobei jedem zur Familie gehörenden Prozeß ein Puffer zugeordnet ist. Da das von uns modellierten System endlich viele Prozesse enthält, und wir voraussetzen, daß die Struktur der Familien von Beginn an festgelegt ist, können wir bei der Modellierung der Puffer mit einer entsprechend definierten Struktur arbeiten. Es gelten:

- Jedem Prozeß Pi der Familie ist Puffer PiB zugeordnet;
- In jedem PiB werden die Nachrichten gespeichert, die andere Prozesse an Pi senden. Da eine 1:1-Zuordnung der Sender und Empfänger vorausgesetzt wird, werden Identifikator und Nachricht gespeichert.
- Jeder PiB enthält die Information darüber, ob von einem bestimmten Prozeß eine Nachricht vorliegt, oder nicht.

Jeder Prozeß kann von jedem der anderen $f-1$ Prozesse der Familie eine Nachricht erhalten. Diese Struktur ist statisch, d.h. die Zahl der möglichen Sender ändert sich nicht. Schließlich setzen wir voraus, daß immer korrekte Werte abgespeichert werden. Die oben beschriebene Struktur läßt sich ähnlich zu der Modellierung der prozeßlokalen Speicherverwalter durch einen *record*-Datentyp beschreiben. Wir betrachten *records*, deren einzelne Elemente jeweils dem Wertebereich $TB_{PiB} = \{Standby, Empty\} \cup Msg$ angehören. Für den Zustandsraum der n Puffer PiB gilt der Datentyp „$StateB_{PiB}$“:

$$State_{PiB} \ : \ \texttt{is record} \ [P1, \ldots, Pf] \ \texttt{of} \ TB_{PiB}$$

Wir setzen voraus, daß diese `records` initialisiert sind. Es gilt für $z \in StateB_{PiB}$:

$$\forall j \in \{1, \ldots, f\} \setminus \{i\} \ : \ z[Pj] := Empty$$

Mit der Abbildung 6.4.1 erhalten wir die ANDL-Spezifikation von Abbildung 6.4.4:

agent PiB
  input channels $V_{F1}toPiB \ : \ S_{V_{F1}toPiB}$
  output channels $KV_{F1}toPi \ : \ S_{KV_{F1}toPi}$
  private channels $\emptyset$
is basic
  $\boldsymbol{f_{PiB}}$ mit der Spezifikation von Seite 149
end PiB

Abbildung 6.4.4: ANDL-Spezifikation von PiB

Für die den Kanälen zugeordneten Nachrichtentypen gilt:

Kanal n	Nachrichtenmengen S_n	
$KV_{F1}toPi$	$\{(val, pid) \mid val \in Msg, pid \in \{P1, \ldots, Pf\} \setminus \{Pi\}\}$	$\cup \ ?!N_{KoopRV}$
$V_{F1}toPiB$	*siehe Tabelle 6.4.1*	

Tabelle 6.4.2: Nachrichtentypen für PiB

Textuell beschreiben wir das Verhalten der PiB wie folgt. $z \in StateB_{PiB}$ sei dabei der interne Speicher eines Puffers, der entsprechend initialisiert ist.

(1) PiB wird durch den Erhalt von $\langle (val, Pj), !ConftoPj \rangle$ oder $Rec(Pj)$ gestartet und geht in den Zustand $z[Pj] = val$ bzw. $z[Pj] = Standby$ über.

(2) PiB erhält über Kanal $V_{F1}toPiB$ die Sequenz $\langle (val, Pj), !ConftoPj \rangle$.

  (a) Es gilt $z[Pj] = Empty$. PiB geht in den Zustand $z[Pj] = val$ über.
  Pi ist nicht empfangsbereit, die gesendete Nachricht wird gespeichert.

  (b) Es gilt $z[Pj] = Standby$. PiB sendet über Kanal $KV_{F1}toPi$ die Sequenz $\langle (val, Pj), !ConftoPj \rangle$ und geht in den Zustand $z[Pj] = Empty$ über.
  Pi ist empfangsbereit, die Nachricht wird an Pi gesendet, und das Rendezvous wird vorbereitet.

(3) PiB erhält über Kanal $V_{F1}toPiB$ die Nachricht $Rec(Pj)$.

(a) Es gilt $z[Pj] = val$. PiB sendet die Sequenz $\langle val, !ConftoPj \rangle$ über Kanal $KV_{F1}toPi$.

Pi führt eine Empfangsoperation aus. Die Nachricht liegt bereits vor, sie wird an Pi gesendet, und das Rendezvous wird vorbereitet.

(b) Es gilt $z[Pj] = Empty$. PiB geht in den Zustand $z[Pj] = Standby$ über.

Pi führt eine Empfangsoperation aus. Die Nachricht liegt nicht vor, die Empfangsbereitschaft von Pi wird registriert.

Die textuelle Beschreibung wird in folgende formale Spezifikation umgesetzt:

Funktionsgleichungen für f_{PiB}

$\forall s \in \prod_{n \in N_{KoopRV}} [S_n^*]$, $val \in Msg$, $z \in StateB_{PiB}$, $Pj \in \{P1, \ldots, Pf\} \setminus \{Pi\}$:
$\exists h \in StateB_{PiB} \rightarrow Type_{PiB}$:

$(1a) \quad f_{PiB}(\{V_{F1}toPiB \mapsto \langle (val, Pj), !ConftoPi \rangle\} \circ s) = h(z[Pj] := val)(s)$

$(1b) \quad f_{PiB}(\{V_{F1}toPiB \mapsto \langle Rec(Pj) \rangle\} \circ s) = h(z[Pj] := Standby)(s)$

wobei in (1a) und (1b) gilt: $\forall Pk \in \{P1, \ldots, Pf\} \setminus \{Pj\} : z[Pk] = Empty$

Für $z[Pj] = Empty$

$(2a) \quad h(z)(\{V_{F1}toPiB \mapsto \langle (val, Pj), !ConftoPj \rangle\} \circ s) = h(z[Pj] := val)(s)$

Für $z[Pj] := Standby$

$(2b) \quad h(z)(\{V_{F1}toPiB \mapsto \langle (val, Pj), !ConftoPj \rangle\} \circ s)$
$= \{KV_{F1}toPi \mapsto \langle val, !ConftoPj \rangle\} \circ h(z[Pj] := Empty)(s)$

Für $z[Pj] \in Msg$

$(3a) \quad h(z)(\{V_{F1}toPiB \mapsto \langle Rec(Pj) \rangle\} \circ s)$
$= \{KV_{F1}toPi \mapsto \langle z[Pj], !ConftoPj \rangle\} \circ h(z[Pj] := Empty)(s)$

Für $z[Pj] = Empty$

$(3b) \quad h(z)(\{V_{F1}toPiB \mapsto \langle Rec(Pj) \rangle\} \circ s)$
$= \{KV_{F1}toPi \mapsto \langle Empty \rangle\} \circ h(z[Pj] := Standby)(s)$

6.4.2 Kommunizierende Prozesse mit Rendezvous

In Abschnitt 6.2 wurde das erweiterte Zustandsdiagramm für Prozesse mit Abbildung 6.2.1 bereits gezeigt. Gemäß dieser Erweiterung der Zustandsmenge werden wir die Spezifikation eines Prozesses Pi anpassen. Der Prozeß führt in Verbindung mit dem Prozessor die

Operationen zum Nachrichtenaustausch durch. Für den Austausch der Empfangsbestätigung, also das *Rendezvous*, führen wir eine neue Kanalverbindung ein, die ausschließlich für die Durchführung des Rendezvous zwischen Sender und Empfänger genutzt wird. Ist die nächste auszuführende Instruktion eine Sendeoperation, so sendet Pi zusätzlich den privaten Port $!ConftoPi$ an den Prozessor. Diesen Fall müssen wir ausschließlich bei der Durchführung eines Rendezvous modellieren. Wir erweitern den Zustandsraum eines Prozesses um den Zustand *waitKoop*:

$$State_{Pi} \; = \; \{ready, busy, waitMem, \boldsymbol{waitKoop}\}$$

Die Zustandsübergänge erfolgen von *ready* zu *busy*, wenn der Prozessor zugeteilt wird, von *busy* zu *ready*, wenn der Prozeß suspendiert wird, und von *busy* zu *waitKoop*, wenn der Prozeß eine Sende- bzw. Empfangsoperation mit einem Rendezvous ausführt und daher blockiert wird, siehe auch Abbildung 6.2.1.

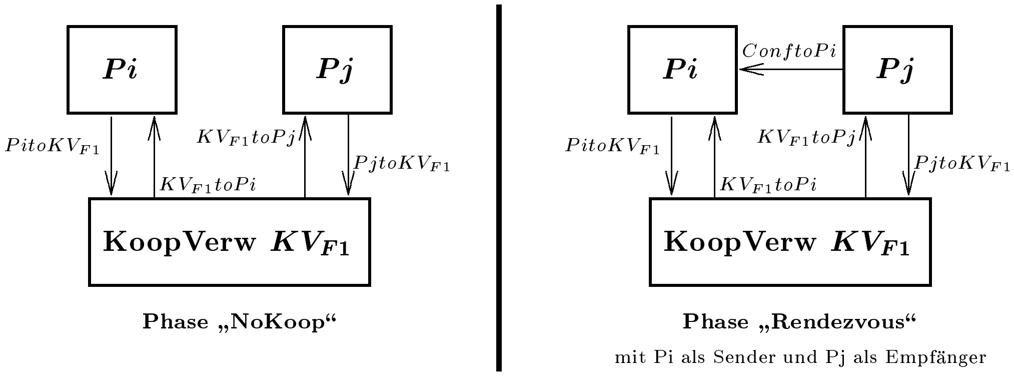


Abbildung 6.4.5: Blockierendes Senden und Empfangen zweier Prozesse

Für die bisher von uns durchgeführten Modellierungen war es ausreichend, die Berechnungsschritte eines Prozesses durch die Menge $STEP$ zu beschreiben. Da wir mit der Erweiterung um das Konzept der Prozeßkooperation spezielle Instruktionen betrachten, werden wir die in Abschnitt 6.1.2 erklärten Operationen `send` und `receive` mit $Send(val, Pj)$ und $Rec(Pj)$ beschreiben und die Menge $STEP$ entsprechend erweitern. Es gilt:

$$STEP_{Koop} \; = \; STEP \; \cup \; \{Send(val, Pj), Rec(Pj) \mid val \in Msg, Pj \in \{P1, \ldots, Pn\}\}$$

Prozesse sind mobile Komponenten, die über einen weiteren privaten Port verfügen. Dieser wird eingesetzt, um die Verbindung zu ihrem Kooperationspartner herzustellen. Über diese Verbindung wird das erforderliche Rendezvous durchgeführt, und ein Prozeß wird aus dem Zustand *waitKoop* gelöst. Abbildung 6.4.5 zeigt die Phasen, die die Prozesse Pi und Pj durchlaufen, wenn Pi eine Sende- und Pj eine Empfangsoperation miteinander ausführen. Auf die Darstellung der Verbindung zum Prozessor verzichten wir hier. Für einen Prozeß Pi erhalten wir die in Abbildung 6.4.6 gezeigte ANDL-Spezifikation.

agent Pi
  input channels $In_i : S_{In_i}$
  output channels $Out_i : S_{Out_i},\ PitoQ : S_{PitoQ}$
  private channels $DtoPi : S_{DtoPi},\ \boldsymbol{ConftoPi\ :\ S_{ConftoPi}}$
is basic
  $\boldsymbol{f_{Pi}}$ mit der Spezifikation von Seite 133 und den Erweiterungen von Seite 152
end Pi

Abbildung 6.4.6: ANDL-Spezifikation von Pi mit Rendezvous

Über den neu eingeführten privaten Kanal $ConftoPi$ der Prozesse werden ausschließlich Bestätigungen über das erfolgreich durchgeführte Rendezvous gesendet. Hierbei wird die Menge N_{KoopRV} der im System gültigen Kanalbezeichner neu definiert. Diese ergibt sich aus der Menge N_{Koop} durch die Hinzunahme der privaten Ports $ConftoPj$ der Prozesse. Wir legen fest: $\forall j \in \{1, \ldots, n\}\ :\ S_{ConftoPj} = \{Conf\}\ \cup\ ?!N_{KoopRV}$

Für die Ausführung von Sende- und Empfangsoperationen wird das Verhalten von Pi erweitert. Hierbei greifen wir auf die in Abschnitt 5.8.1 gezeigte Modellierung des Prozesses zurück. Dort wurde beschrieben, welches Verhalten für Pi gefordert ist, wenn ein Seitenfehler auftritt. Analog zur Modellierung eines Prozesses mit Speicherfähigkeit geben wir zusätzlich die Zugriffsrechte an den Kanalverbindungen zum Kooperationsverwalter an den Prozessor weiter, wenn er einem Prozeß zugeteilt wird. Im folgenden wird das Verhalten um die Fälle erweitert, in denen ein Nachrichtenaustausch stattfindet und der Prozeß blockiert wird, bis das Rendezvous ausgeführt wird. Es gilt $p \in STEP^*_{Koop}$.

(1") Erhält Pi im Zustand *ready* über Kanal $DtoPi$ die Ports $?PZtoP$ und $!PtoPZ$, so sendet Pi über Kanal $PtoPZ$ die nächste Instruktion, die Ports $?SVtoPi$ und $PitoSV$ **sowie die Ports $\boldsymbol{?KV_{F1}toPi}$ und $\boldsymbol{!PitoKV_{F1}}$** und geht in den Zustand *busy* über.

(3') Erhält Pi im Zustand *busy* die Nachricht $Ok(ft.p)$ über Kanal $PZtoP$ und gilt $\boldsymbol{ft.rt.p = (Send(var, Pj), va)}$, so sendet er die Nachricht $\boldsymbol{Send(var, Pj)}$ **und den Port $\boldsymbol{!ConftoPi}$** über Kanal $PtoPZ$ und verbleibt im Zustand *busy*.

(4") Erhält Pi im Zustand *busy* über Kanal $PZtoP$ die Sequenz

$$\langle\, Ok(ft.p), Suspend, ?SVtoPi, !PitoSV, \boldsymbol{?KV_{F1}toPi, !PitoKV_{F1}}\,\rangle$$

werden die Verbindungen zum Prozessor gelöscht. Pi bewirbt sich erneut um den Prozessor und geht in den Zustand *ready* über.

(8) Erhält Pi im Zustand *busy* über Kanal $PZtoP$ die Nachricht *Blocked* und die Ports $?SVtoPi$, $!PitoSV$ **sowie $\boldsymbol{?KV_{F1}toPi}$ und $\boldsymbol{!PitoKV_{F1}}$**, wird die Verbindung zum Prozessor gelöst, und Pi geht in den Zustand *waitKoop* über.

(9) Erhält Pi im Zustand *waitKoop* über Kanal $ConftoPi$ die Nachricht $Conf$ und den Port $!ConftoPi$, so sendet er die Sequenz $\langle Pi, !DtoPi \rangle$ über Kanal $PitoQ$ und geht in den Zustand *ready* über.

(10) Erhält Pi im Zustand $(WaitKoop, (Rec(Pj), va) \circ p)$ über Kanal $KV_{F1}toPi$ die Sequenz $\langle m, !ConftoPj \rangle$, so sendet Pi die Sequenz $\langle Conf, !ConftoPj \rangle$ über Kanal $ConftoPj$ sowie die Sequenz $\langle Pi, !DtoPi \rangle$ über Kanal $PitoQ$ und geht in den Zustand $(ready, Write(m, va) \circ p)$ über.

Erweiterung der Spezifikation für f_{Pi} von Seite 133

(1″) $h(ready,\ p)(\{DtoPi \mapsto \langle ?PZtoP, !PtoPZ \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle ft.p, \texttt{chanlist} \rangle\} \circ h(busy, p)(s)$

Für $ft.rt.p = (Send(var, Pj), va)$

(3′) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p) \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle ft.rt.p, \boldsymbol{!ConftoPi} \rangle\} \circ h(busy, rt.p)(s)$

(4″) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p), Suspend, \texttt{chanlist} \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle ?PZtoP, !PtoPZ \rangle,$
$PitoQ \mapsto \langle Pi, !DtoPi \rangle\} \circ h(ready, rt.p)(s)$

(8) $h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p), Blocked, \texttt{chanlist} \rangle\} \circ s)$
$= \{PtoPZ \mapsto \langle ?PZtoP, !PtoPZ \rangle\} \circ h(waitKoop, p)$

Falls $ft.p = (Send(val, Pj), va)$

(9) $h(waitKoop, p)(\{ConftoPi \mapsto \langle Conf, !ConftoPi \rangle\} \circ s)$
$= \{PitoQ \mapsto \langle Pi, !DtoPi \rangle\} \circ h(ready, rt.p)(s)$

(10) $h(waitKoop, Rec(Pj) \circ p)(\{KV_{F1}toPi \mapsto \langle m, !ConftoPi \rangle\} \circ s)$
$= \{ConftoPi \mapsto \langle Conf, !ConftoPi \rangle,$
$PitoQ \mapsto \langle Pi, !DtoPi \rangle\} \circ h(ready, Write(m, va) \circ p)(s)$

wobei : $\texttt{chanlist} = ?SVtoPi, !PitoSV, \boldsymbol{?KV_{F1}toPi, !PitoKV_{F1}}$

Wird eine Sendeoperation ausgeführt, so gibt Pi den Port $!ConftoPi$ an den Prozessor, damit dieser die zu sendende Nachricht sowie den Port für die Ausführung des Rendezvous an den Kooperationsverwalter weitergeben kann. Pi wird blockiert, bis das Rendezvous über Kanal $ConftoPi$ ausgeführt ist, und der Prozessor wird freigegeben. Im Fall eines Seitenfehlers werden die Ports zum Kooperationsverwalter und, falls der Seitenfehler bei der Durchführung einer Sendeoperation auftritt, zusätzlich der Port $!ConftoPi$ zurückgegeben. Diese Erweiterung der Spezifikation wurde hier nicht explizit demonstriert. Punkt (6) der

Spezifikation eines Prozesses von Seite 133 muß entsprechend um den Port $!ConftoPi$ sowie die Ports zu KV_{F1} erweitert werden.

6.5 Nichtblockierendes Senden

Ausgehend von Abschnitt 6.4 beschreiben wir im folgenden die Ausführung der Operationen mit *nichtblockierendem* Senden. Analog zu Abschnitt 6.4 werden wir einen Kooperationsverwalter modellieren, der die sendenden und empfangenden Prozessen koordiniert.

6.5.1 Der Kooperationsverwalter bei nichtblockierendem Senden

Der Kooperationsverwalter wird mit dem strukturellen Aufbau modelliert, den wir bereits für den Nachrichtenaustausch mit Rendezvous verwendet haben. Damit können wir das in Abbildung 6.4.1 gezeigte SSD und die ANDL-Spezifikation von Abbildung 6.4.2 vollständig übernehmen. Der wesentliche Unterschied in den Modellierungen besteht darin, daß keine Ports an den Manager weitergegeben werden und dementsprechend keine Ports an Partnerprozesse weiterzuleiten sind. Wir können die Spezifikationen der **Komponente $\boldsymbol{V_{F1}}$** fast vollständig übernehmen. Die Sequenzen $\langle\, Send(val, Pj), !ConftoPj\,\rangle$ bzw. $\langle\, (m, Pj), !ConftoPj\,\rangle$ müssen abgeändert werden zu: $\langle\, Send(val, Pj)\,\rangle$ bzw. $\langle\, (m, Pj)\,\rangle$. Wir verzichten auf die Spezifikation, die sich gemäß Abschnitt 6.4.1.1 leicht ergibt.

Entsprechend zur Änderung an der Modellierung der Verteilerkomponente sind geringe Änderungen an der Spezifikation der **Nachrichtenpuffer $\boldsymbol{PiB}$** vorzunehmen. Die in Abbildung 6.4.4 gezeigte ANDL-Spezifikation einer Komponente PiB kann übernommen werden. Es ist zu beachten, daß die Kanäle $ConftoPi$ für das System mit nichtblockierendem Senden nicht definiert sind. Wir verwenden somit die Menge der Kanalbezeichner N_{Koop}. Bei nichtblockierendem Senden kann für einen Empfänger eine Liste von Nachrichten vorliegen. Ein Empfänger entnimmt die Nachrichten elementweise, alle gesendeten Nachrichten müssen gespeichert und dürfen nicht überschrieben werden. Dementsprechend erweitern wir den Zustandsraum zu $StateNB_{PiB}$ wie folgt:

$$StateNB_{PiB} \;:\; \texttt{is record}\; [P1, \ldots, Pf] \;\texttt{of}\; TNB_{PiB}$$

mit $TNB_{PiB} = \{Standby, Empty\} \cup \boldsymbol{Msg}^*$.

Das Verhalten von PiB wird textuell beschrieben durch:

(1) **(a)** Erhält PiB die Nachricht (val, Pj) über Kanal $V_{F1}toPiB$, und ist Pi nicht empfangsbereit, wird val abgespeichert. Hierbei werden die Fälle $z[Pj] = Empty$ und $z[Pj] = mstr$ mit $mstr \in Msg^* \setminus \{\langle\rangle\}$ unterschieden.

(b) Erhält PiB die Nachricht (val, Pj) über Kanal $V_{F1}toPiB$, und ist Pi empfangsbereit, wird über Kanal $KV_{F1}toPi$ die Nachricht val gesendet.

(2) **(a)** Erhält PiB die Nachricht $Rec(Pj)$ über Kanal $V_{F1}toPiB$, und liegt von Pj eine Nachricht vor, wird diese über Kanal $KV_{F1}toPi$ gesendet. Hierbei werden die Fälle unterschieden, ob von Pj nur *eine* oder bereits *mehrere* Nachrichten vorliegen.

(b) Erhält PiB die Nachricht $Rec(Pj)$ über Kanal $V_{F1}toPiB$ und liegt von Pj keine Nachricht vor, wird registriert, daß Pj empfangsbereit ist.

Die Umsetzung in die formale Spezifikation ergibt folgende Formalisierung:

Funktionsgleichungen für f_{PiB}

$$\forall s \in \prod_{n \in N_{Koop}} [S_n^*]\ ,\ val \in Msg,\ Pj \in \{P1, \ldots, Pf\} \setminus \{Pi\}\ :\ \exists h \in StateNB_{PiB} \ \rightarrow\ Type_{PiB}\ :$$

Für $z[Pj] = Empty$

(1a.1) $h(z)(\{V_{F1}toPiB \mapsto \langle (val, Pj) \rangle\} \circ s) \ = \ h(z[Pj] := val)(s)$

Für $z[Pj] = mstr$ mit $mstr \in Msg^* \setminus \{\langle\rangle\}$

(1a.2) $h(z)(\{V_{F1}toPiB \mapsto \langle (val, Pj) \rangle\} \circ s) \ = \ h(z[Pj] := z[Pj] \circ val)(s)$

Für $z[Pj] = Standby$

(1b) $h(z)(\{V_{F1}toPiB \mapsto \langle (val, Pj) \rangle\} \circ s) \ = \ \{KV_{F1}toPi \mapsto \langle val \rangle\} \circ h(z[Pj] := Empty)(s)$

Für $z[Pj] = m$ mit $m \in Msg$

(2a.1) $h(z)(\{V_{F1}toPiB \mapsto \langle Rec(Pj) \rangle\} \circ s) \ = \ \{KV_{F1}toPi \mapsto \langle z[Pj] \rangle\} \circ h(z[Pj] := Empty)(s)$

Für $z[Pj] = mstr$ mit $mstr \in Msg^* \setminus \{\langle\rangle\}$

(2a.2) $h(z)(\{V_{F1}toPiB \mapsto \langle Rec(Pj) \rangle\} \circ s) \ = \ \{KV_{F1}toPi \mapsto \langle ft.z[Pj] \rangle\} \circ h(z[Pj] := rt.z[Pj])(s)$

Für $z[Pj] = Empty$

(2b) $h(z)(\{V_{F1}toPiB \mapsto \langle Rec(Pj) \rangle\} \circ s) \ = \ h(z[Pj] := Standby)(s)$

6.5.2 Prozesse mit nichtblockierendem Senden

Bei dieser Form des Nachrichtenaustauschs führen die Partner kein Rendezvous aus. Der Sender liefert seine Nachricht beim Kooperationsverwalter ab und setzt die Durchführung seiner Berechnung fort. Die in Abschnitt 6.4.2 festgelegte Erweiterung des Zustandsraums $State_{Pi}$ um den Zustand $waitKoop$ bleibt jedoch auch bei nichtblockierendem Senden bestehen. Führt ein Prozeß Pi eine Empfangsoperation aus und liegt die erwartete Nachricht noch nicht vor, so wird Pi blockiert. Der Kooperationsverwalter sorgt dafür, daß ein Prozeß entblockiert wird, sobald der sendende Partner die Nachricht abgeliefert hat.

Ausgehend von der in Abschnitt 6.4.2 gezeigten Modellierung des Verhaltens leiten wir das Verhalten für nichtblockierendes Senden ab. Die Erweiterung der Spezifikation eines Prozesses für nichtblockierendes Senden können wir mit nur geringfügigen Änderungen übernehmen. Die Punkte (1”), (4”) und (8) können vollständig und unverändert wiederverwendet werden. Schritt (3’) dessen Anpassung in Abschnitt 6.4.2 nötig war, da Pi die Vorbereitungen für das Rendezvous treffen muß, ist hier nicht erforderlich, er tritt in der Spezifikation nicht mehr auf. Ebenso ist die Spezifikation von Punkt (9) nicht nötig, da ein Prozeß bei der Ausführung einer Sendeoperation nicht blockiert wird und nicht auf die Bestätigung des Empfängers wartet. Punkte (10) wird wie folgt angepaßt und formalisiert:

(10’) Erhält Pi im Zustand $(waitKoop, (Rec(Pj), va) \circ p)$ über Kanal $KV_{F1}toPi$ die Nachricht $m \in Msg$, sendet Pi die Sequenz $\langle Pi, !DtoPi \rangle$ über Kanal $PitoQ$ und geht in den Zustand $(ready, Write(m, va) \circ p)$ über.

Anpassung der Spezifikation für f_{Pi} der Seiten 133 und 152

$$
\begin{aligned}
(10') \quad & h(waitKoop, (Rec(Pj), va) \circ p)(\{KV_{F1}toPi \mapsto \langle m \rangle\} \circ s) \\
& \quad = \{PitoQ \mapsto \langle Pi, !DtoPi \rangle\} \circ h(ready, Write(m, va) \circ rt.p)(s)
\end{aligned}
$$

6.6 Erweiterung des Prozessors

Mit der Modellierung des Nachrichtenaustauschs wurden die Berechnungsschritte bereits bei den Prozessen detaillierter beschrieben. Zusätzlich verfügt der Prozessor über den Kooperationsverwalter des Prozesses und initiiert in Verbindung mit diesem den Nachrichtenaustausch zwischen Prozessen. Wir erweitern die in Abschnitt 5.8.2 gezeigte Modellierung des Prozessors. Dessen Verhalten wurde so beschrieben, daß auf dem Arbeitsspeicher abstrakte Speicheroperationen $Step_i$ mit realer Adresse ra_i ausgeführt werden. Mit einer Sendeoperation wird ein Wert an den Partnerprozeß gesendet, der aus dem Speicher entnommen wird. Gemäß dieser Vorstellung wird die Spezifikation des Prozessors so erweitert, daß er bei einer Nachricht $(Send(val, Pj), va)$ und bei nicht vorliegendem Seitenfehler einen Wert aus dem Speicher liest. Dieser Wert wird an den Kooperationsverwalter gesendet. Die Modellierung des Prozessors wird ebenfalls an die beiden alternativen Paradigmen des Nachrichtenaustauschs angepaßt. Wir zeigen beide in der folgenden Spezifikation.

Zusätzlich ist zu beachten, daß ein Prozeß während der Ausführung einer Instruktion suspendiert werden kann. In Abschnitt 5.8.2 haben wir bereits detailliert die verschiedenen Fälle spezifiziert, die hierbei auftreten können. Die Ausführung einer Instruktion ist im Gegensatz zur Modellierung im Kapitel 4 in mehrere Einzelschritte aufgeteilt. Bei der Ausführung jedes dieser Schritte kann die Nachricht *Suspend* über die Verbindung zum Dispatcher eintreffen. Vor allem bei der Ausführung einer Sendeoperation sind hier bei der Modellierung mehrere zusätzliche Fallunterscheidungen nötig. Diese werden wir im folgenden nicht demonstrieren. Wir verweisen hierfür auf die Spezifikation des Prozessors in Abschnitt 5.8.2 und zeigen die Spezifikation nur für den Fall, daß der Nachrichtenaustausch vollständig durchgeführt werden kann, ohne daß der Prozeß suspendiert wird. PZ sei aktuell mit einem Prozeß Pi verbunden, der zur Familie $F1$ gehört.

(1) Erhält PZ über Kanal $PtoPZ$ die Nachricht $(Rec(Pj), va)$ und die Sequenz $v = \langle\rangle$ über Kanal $DtoPZ$, so sendet PZ die Nachricht $Rec(Pj)$ über Kanal $PitoKV_{F1}$ und über Kanal $PZtoP$ die Nachricht $Blocked$ sowie die Ports $?SVtoPi$, $!PitoSV$, $?KV_{F1}toPi$ und $!PitoKV_{Fk}$ über Kanal $PZtoP$.

(2) Erhält PZ im Zustand $(Send(val, Pj), va)$ über Kanal $SVtoPi$ die Nachricht (va, ra), sendet PZ die Nachricht $Read(ra)$ über Kanal $PZtoAS$.

(3) **(a)** *Nichtblockierendes Senden*:
Erhält PZ im Zustand $(Send(val, Pj), va)$ über Kanal $AStoPZ$ die Nachricht $m \in Msg$, so sendet er die Nachrichten $Send(m, Pj)$ über Kanal $PitoKV_{F1}$ und $Ok((Send(val, Pj), va)$ über Kanal $PZtoP$.

(b) *Blockierendes Senden*:
Erhält PZ im Zustand $(Send(val, Pj), va)$ über Kanal $AStoPZ$ die Nachricht $m \in Msg$, so sendet er die Nachricht $Send(m, Pj)$ und den Port $!ConftoPi$ über Kanal $PitoKV_{F1}$ und $Blocked$ sowie die Ports $?SVtoPi$, $!PitoSV$, $?KV_{F1}toPi$ und $!PitoKV_{Fk}$ über Kanal $PZtoP$

In Punkt (1) müssen wir nicht zwischen blockierendem und nichtblockierendem Senden unterscheiden, da Prozesse bei der Ausführung einer Empfangsoperation stets blockieren.

Erweiterung der Spezifikation für f_{PZ} von Seite 136

(1) $$f_{PZ}(\{PtoPZ \mapsto \langle (Rec(Pj), va) \rangle, DtoPZ \mapsto \langle\rangle\} \circ s) = \{PitoKV_{F1} \mapsto \langle Rec(Pj) \rangle, PZtoP \mapsto \langle Blocked, \mathtt{chanlist} \rangle\} \circ f_{PZ}(s)$$

(2) $$g((Send(val, Pj), va))(\{SVtoPi \mapsto \langle (va, ra) \rangle\} \circ s) = \{PZtoAS \mapsto \langle Read(ra) \rangle\} \circ g(Send(val, Pj), va))(s)$$

Für nichtblockierendes Senden:

(3a) $$g((Send(val, Pj), va))(\{AStoPZ \mapsto \langle m \rangle\} \circ s) = \{PitoKV_{F1} \mapsto \langle Send(m, Pj) \rangle, PZtoP \mapsto \langle Ok(Send(val, Pj), va)\} \circ f_{PZ}(s)$$

Für blockierendes Senden:

(3b) $$g((Send(val, Pj), va))(\{AStoPZ \mapsto \langle m \rangle\} \circ s) = \{PitoKV_{F1} \mapsto \langle Send(m, Pj) \rangle, PZtoP \mapsto \langle Blocked, \mathtt{chanlist} \rangle\} \circ f_{PZ}(s)$$

wobei : $\mathtt{chanlist} = ?SVtoPi, !PitoSV, ?KV_{F1}toPi, !PitoKV_{F1}$

Kapitel 7

Prozeßverwaltung

Die bisher entwickelte Modellierung behandelt wesentliche Bereiche der Ressourcenverwaltung auf hohem Abstraktionsniveau. Berechnungen werden durch die Prozesse schrittweise ausgeführt. Um diese Ausführung voranschreiten zu lassen, werden Prozessor und Speicher benötigt. Ein Prozeß wird suspendiert, wenn die Zeitspanne, für die ihm der Prozessor zugeteilt war, abgelaufen ist, siehe Kapitel 4. Er wird blockiert, wenn ein Seitenfehler auftritt, und die Speicherverwaltung sorgt dafür, daß die Ursache für den Seitenfehler behoben wird, siehe Kapitel 5. Mit der Prozeßkooperation aus Kapitel 6 können Prozesse, die derselben Familie angehören, Nachrichten austauschen und auf diese Weise kooperieren.

Für das schrittweise entwickelte System haben wir stets vorausgesetzt, daß eine feste, statische Anzahl von n Prozessen verwaltet wird. Prozesse wurden bisher *nicht* erzeugt. Wir sind durchgängig davon ausgegangen, daß die Verwaltungskomponenten über die notwendigen Informationen verfügen. Ein Beispiel hierfür ist die Zuordnung von Seiten zu Blöcken und Seitenrahmen innerhalb eines lokalen Speicherverwalters. In diesem Kapitel wird die Modellierung des Systems vervollständigt, indem Prozesse auf der Basis von Benutzeraufträgen erzeugt und mit dem Abschluß der Berechnung aufgelöst werden. Wir erweitern das System um einen eigenständigen Bereich, mit dem die Prozeßerzeugung modelliert wird. Von der Erweiterung des Systems ist vor allem die Speicherverwaltung betroffen, da ein Prozeß nur dann erzeugt werden kann, wenn genügend Speicherplatz vorhanden ist.

7.1 Einführung

Die Aufträge, die der Benutzer an ein Rechensystem stellt, beschreiben Berechnungen, die er unter Ausnutzung der Rechenleistung des Systems ausführen möchte. Die Schritte, die hierbei durchgeführt werden, werden als Programm (oder Benutzerauftrag) an das Rechensystem gegeben. Um das Programm ausführen zu können, muß es in maschinennaher Sprache vorliegen. In dieser Form ist der Prozessor in der Lage, die einzelnen Befehle direkt auszuführen. Da ein Benutzerauftrag üblicherweise nicht in dieser Sprache formuliert ist, muß er entsprechend übersetzt werden. Die Übersetzung eines Programms in einen maschinensprachlichen Code durch einen *Compiler* resultiert aus der Schnittstellenfunktion zwischen Benutzer und Rechensystem, für die ein Betriebssystem ebenfalls zuständig ist; vergleiche Kapitel 8.1 in [GS94] oder Abschnitt 4.1.3 in [Spi95].

Ein Prozeß wird zur Ausführung eines auf diese Weise hergeleiteten Programms erzeugt. Er terminiert, wenn die Ausführung der Berechnung abgeschlossen ist. Da für einen Prozeß Speicherplatz benötigt wird, kann er erst dann erzeugt werden, wenn freier Speicherplatz in ausreichender Größe zur Verfügung steht. Terminiert der Prozeß wird der von ihm belegte Speicherplatz freigegeben. Um die Berechnung ausführen zu können, muß ein Prozeß bei seiner Erzeugung so in das bestehende System integriert werden, daß er den Prozessor und alle weiteren Betriebsmittel in Verbindung mit den Verwaltungskomponenten des Betriebssystems nutzen kann. Diese Integration erfolgt durch die Speicher- und Prozeßverwaltung. Da ein Prozeß einer *Familie* angehört, mit deren Mitgliedern er kooperieren kann, muß dafür gesorgt werden, daß er gemäß seiner Familienzugehörigkeit in die Menge der bereits existierenden Prozesse eingegliedert wird.

Die Maßnahmen zur Integration eines Prozesse in ein bestehendes System sind in der Literatur bei der Beschreibung des Prozeßkonzepts aufgeführt. Im Detail werden Prozeßerzeugung und Terminierung beschrieben, siehe beispielsweise die Kapitel 3 in [Sta92], 4.3.1 und 4.3.2 in [GS94] oder 4.1.3 in [Spi95]. Der Zusammenschluß von Prozessen zu Teilmengen, den Familien, ist eine Möglichkeit zur Beschreibung von Abhängigkeiten zwischen Prozessen, die insbesondere bei der Erzeugung von Kindprozessen entstehen; vergleiche hierzu ebenfalls die oben genannten Abschnitte zur Erzeugung von Prozessen. Für ein Rechensystem muß sichergestellt sein, daß das System nur von Benutzern, die über eine Zugangsberechtigung zum System verfügen, benutzt wird. Die Hinzunahme eines Pförtners (*Login*) zum modellierten System ist eine stark vereinfachende Maßnahme zu den in Kapitel 14 in [GS94] oder Kapitel 10.2 in [Sta92] beschriebenen Verfahren zur Sicherheit in Rechensystemen.

7.2 Methodische Vorgehensweise

Alle bisher gezeigten Modellierungen basieren darauf, daß für das System eine feste, statische Anzahl von n Prozessen festgelegt ist. Das Zustandsdiagramm, das mit den Modellierungen von Kapitel 6 gültig ist, umfaßt bereits die, bezogen auf die Betriebsmittelverwaltung, wesentlichen Zustände für Prozesse. Zur Vervollständigung des Verhaltens eines Prozesses und des von uns entwickelten Systems nehmen wir die Zustände *started* und *terminated* hinzu und erhalten das Zustandsdiagramm aus Abbildung 7.2.1.

Mit den in diesem Kapitel vorgenommenen Erweiterungen wird ein Prozeß auf der Basis eines gültigen Benutzerauftrags erzeugt, den das System von seiner Umgebung erhält. Es wird vorausgesetzt, daß ein neuer Prozeß nur dann erzeugt werden kann, wenn genügend freier Speicherplatz verfügbar ist – in unserem Fall freie Seitenrahmen des Arbeitsspeichers. Wir erhalten als Aufgabenstellung für die abschließende Erweiterung des Systems:

> Für ein Einprozessorsystem werden neben der Verwaltung der Betriebsmittel „Prozessor" und „Speicher" sowie den kooperierenden Prozessen das Erzeugen und Auflösen von Prozessen sowie ein Pförtner zum System modelliert.

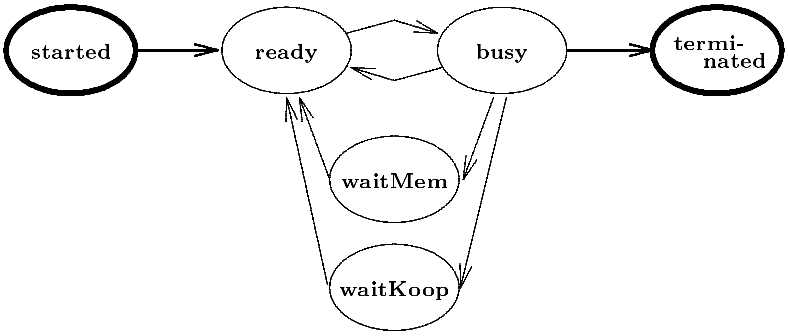


Abbildung 7.2.1: Das vollständige Zustandsdiagramm für Prozesse

Das System, auf dem unsere abschließende Erweiterung basiert, besteht aus den Teilsystemen, die für die Prozessor- und Speicherverwaltung sowie die Koordination des Nachrichtenaustauschs zuständig sind. Wir gehen davon aus, daß diese Teilsysteme und alle zentralen Verwaltungskomponenten im System enthalten sind. Sie erhalten die benötigten Informationen über einen neuen Prozeß bei dessen Erzeugung. Dies gilt nicht für die lokalen Speicherverwalter, die Prozessen jeweils direkt zugeordnet sind. In unserer Modellierung werden sie vom globalen Speicherverwalter erzeugt und mit der für den neuen Prozeß gültigen Information bzgl. dessen Realisierung in Arbeits- und Hintergrundspeicher initialisiert. Wir gehen davon aus, daß im System m Prozeßfamilien und damit m Kooperationsverwalter enthalten sind. Ein neuer Prozeß wird in *seine* Familie eingebunden. Die Information hierzu wird aus der Beschreibung des Benutzerauftrags hergeleitet.

Für die Erzeugung eines Prozesses ist ein weiteres Teilsystem, die *Prozeßverwaltung*, zuständig, das gültige Benutzeraufträge entgegennimmt. Dieses Subsystem ist mit den bereits spezifizierten Teilsystemen verbunden und übernimmt die Koordination einer Prozeßerzeugung. Die Zugangsberechtigung eines Benutzers wird durch eine separate Komponente, den *Pförtner*, überprüft; er läßt ausschließlich Aufträge von berechtigten Benutzern für das System zu. Abbildung 7.2.2 zeigt die Grobstruktur des zu entwickelnden Systems.

Für die neu zu entwickelnden und bereits spezifizierten Komponenten werden folgende Anpassungen und Erweiterungen vorgenommen:

Pförtner: Der Pförtner überprüft die Zugangsberechtigung eines Benutzers. Nur Aufträge von berechtigten Benutzern werden für das System zugelassen. Der Pförtner wird vollständig neu spezifiziert und in das System eingebunden.

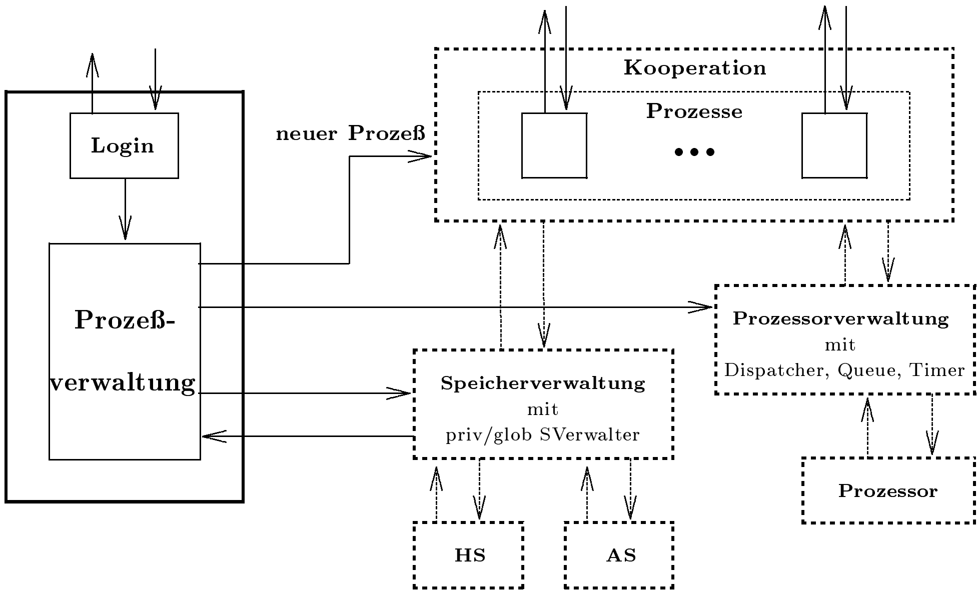


Abbildung 7.2.2: Erweitertes System mit Prozeßerzeugung

Pförtnerqueue: Alle Benutzeraufträge, die die Überprüfung durch den Pförtner bestanden haben, werden gespeichert. Die Queue wird als neue Komponente zum System hinzugenommen, wobei wir auf bereits spezifizierte Queues zurückgreifen.

Prozeßverwalter: Zur Ausführung der Aufträge berechtigter Benutzer erzeugt der Prozeßverwalter Prozesse und integriert sie in das System. Aufgrund der in jedem Auftrag codierten Informationen bzgl. des erforderlichen Speicherplatzes und der Einordnung in eine Familie kann die Erzeugung erfolgen. Der Prozeßverwalter überprüft mit Hilfe des globalen Speicherverwalters, ob freier Speicherplatz in ausreichender Größe verfügbar ist. Ist dies der Fall, wird ein Prozeß erzeugt und in seine Familie eingeordnet. Der Prozeßverwalter wird vollständig neu spezifiziert.

Prozesse: werden bei Bedarf und zur Ausführung eines Benutzerauftrags erzeugt. Die Modellierung der Prozesse wird übernommen und geringfügig ergänzt: Ein Prozeß, dessen Berechnung terminiert, wird aufgelöst.

globaler Speicherverwalter: Ein Prozeß kann nur erzeugt werden, wenn freier Speicherplatz der geforderten Größe und dabei insbesondere genügend freie Seitenrahmen verfügbar sind. Vom Prozeßverwalter erhält der globale Speicherverwalter die Information über die Anzahl der benötigten Seitenrahmen. Die bereits erstellte Modellierung wird so erweitert, daß über belegte und freie Seitenrahmen und Blöcke Buch geführt wird. Kann ein Prozeß auf dieser Basis erzeugt werden, wird Speicherplatz belegt, und ein lokaler Speicherverwalter wird erzeugt. Von privaten Speicherverwaltern werden die Meldungen über freigewordene Speicherbereiche empfangen; diese müssen registriert werden. Die Modellierung des globalen Speicherverwalters wird angepaßt und erweitert.

lokale Speicherverwalter: werden erzeugt, sobald für einen neuen Prozeß Speicherplatz belegt wird. Wird ein Prozeß aufgelöst, werden die frei gewordenen Speicherbereiche dem globalen Speicherverwalter zurückgegeben, und der lokale Speicherverwalter wird aufgelöst. Die Modellierung wird entsprechend angepaßt.

Kooperationsverwalter: erhalten vom Prozeßverwalter die Information, daß ein Prozeß neu erzeugt und in die zugehörige Familie integriert wurde. Dies erfolgt durch das Versenden von Ports. Weitere Anpassungen der Modellierung sind nicht erforderlich.

Für die hier nicht genannten, bereits modellierten Komponenten sind nur geringfügige Anpassungen vorzunehmen. Die Queue, mit der das Scheduling modelliert wird, erhält einen weiteren Kanal und ist in der Lage, alle im System gültigen Prozeßidentifikatoren abzuspeichern. Alle weiteren Komponenten, wie Dispatcher, Timer und Prozessor oder alle Prozesse und deren lokale Verwaltungskomponenten sind von der erweiterten Funktionalität ebenfalls nur indirekt betroffen. Ihre Spezifikationen müssen nicht oder nur minimal erweitert werden. Diese einfachen Erweiterungen werden wir nicht explizit durchführen.

7.3 Das erweiterte System mit Prozeßerzeugung

Aus der in Abschnitt 7.2 vorgegebenen Aufgabenstellung und mit den ersten Erläuterungen zur Modellierung ergibt sich ein verteiltes Focus-System mit Prozessen und den Komponenten zur Prozessor- sowie Speicherverwaltung. Die Prozesse sind zu Familien zusammengeschlossen und können mit den Prozessen ihrer Familie mittels Nachrichtenaustausch kooperieren. Zu diesem System werden die Anteile hinzugefügt, die die Prozeßerzeugung und den Pförtner als Schnittstelle zur Umgebung enthalten. Der Prozeßverwalter ist mit allen Komponenten verbunden, die von der Integration eines neuen Prozesses betroffen sind. Der Pförtner ist direkt mit der Umgebung des modellierten Systems und mit dem Prozeßverwalter indirekt über eine Queue verbunden.

Das System `BS` durchläuft, bezogen auf einen neuen Prozeß Pi, zwei Phasen: Zunächst ist Pi noch nicht im System eingebunden. Nach seiner Erzeugung gemäß dem bereits erläuterten Verhalten sind sowohl Pi als auch der lokale Speicherverwalter SV_{Pi} im System integriert. Abbildung 7.3.1 zeigt die beiden Phasen des Systems, wobei wir davon ausgehen, daß Pi in die Familie F_k eingeordnet wird. Zur Steigerung der Übersichtlichkeit verzichten wir auf die graphische Darstellung der Komponenten, die von der Erzeugung eines neuen Prozesses nicht direkt betroffen sind: alle weiteren Prozesse sowie deren Speicher- und Kooperationsverwalter, die Speicher AS und HS sowie Prozessor, Dispatcher und Timer.

Ausgehend vom in Abbildung 7.3.1 gezeigten SSD leiten wir die in Abbildung 5.4.2 gezeigte Andl-Spezifikation her. In der initialen Phase sind im von uns modellierten System kein Prozeß, aber alle globalen Verwaltungskomponenten enthalten. Wir verzichten auf die Nennung der Kanäle und Komponenten, die in den vorangegangenen Kapiteln entwickelt wurden und die von der abschließenden Erweiterung des Systems nur indirekt betroffen

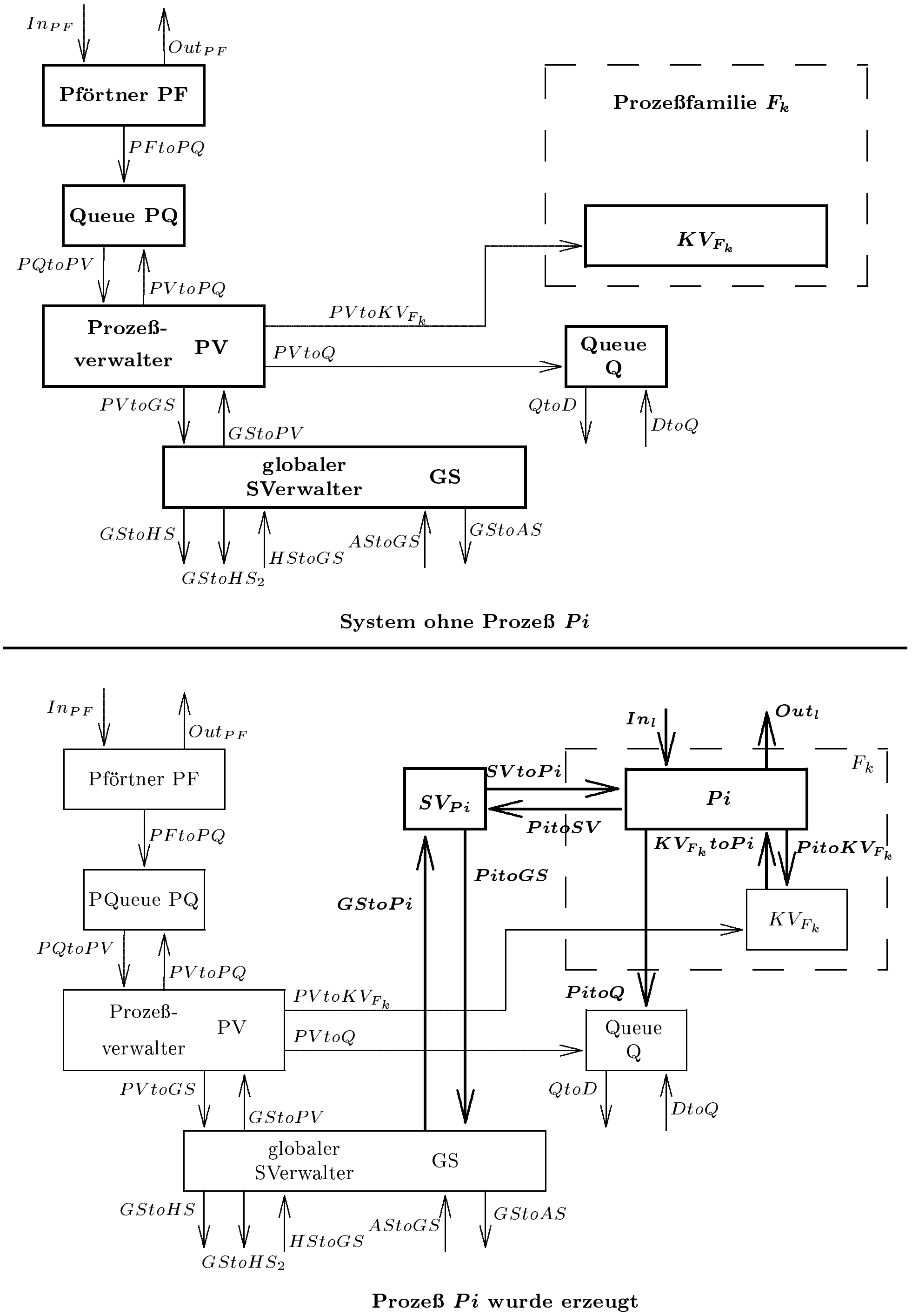


Abbildung 7.3.1: Erzeugung und Integration eines Prozesses Pi

sind. Die Nachrichtentypen für die neu eingeführten Kanäle werden in den folgenden Abschnitten erläutert und definiert. Die Menge der für das System gültigen Kanalbezeichner sei durch die Menge N_{BS} definiert, wobei wir zur Definition der Menge privater Ports $N_{private}$ auf die im folgenden entwickelten Spezifikation von Pförtner, Prozeßverwalter und globalem Speicherverwalter verweisen. $N_{private}$ enthält alle privaten Ports, die für das System definiert sein müssen, um Prozesse und lokale Speicherverwalter zu erzeugen und in das bestehende System zu integrieren.

agent BS
  input channels $In_{PF} : S_{In_{PF}}$
  output channels $Out_{PF} : S_{Out_{PF}}$
is network

$$
\begin{array}{lcl}
\ll Out_{PF} \gg & = & \mathtt{PF} \ll In_{PF} \gg \; ; \\
\ll PQtoPV \gg & = & \mathtt{PQ} \ll PFtoPQ, PVtoPQ \gg \; ; \\
\ll PVtoPQ, PVtoQ, PVtoGS, & & \\
\quad PVtoKV_{F_1}, \ldots, PVtoKV_{F_m} \gg & = & \mathtt{PV} \ll PQtoPV, GStoPV \gg \; ; \\
\ll\gg & = & \mathtt{KV}_{F1} \ll PVtoKV_{F_1} \gg \; ; \\
\vdots & \vdots & \vdots \\
\ll\gg & = & \mathtt{KV}_{Fm} \ll PVtoKV_{F_m} \gg \; ; \\
\ll GStoPV, GStoAS, & & \\
\quad GStoHS, GStoHS_2 \gg & = & \mathtt{GS} \ll PVtoGS, AStoGS, HStoGS \gg \; ; \\
\vdots & \vdots & \vdots
\end{array}
$$

$$[sowie\ die\ Komponenten: Dispatcher,\ Queue,\ Timer,\ Prozessor, Hintergrund-\ und\ Arbeitsspeicher]$$

end BS

Abbildung 7.3.2: ANDL-Spezifikation für das Netz mit Prozeßverwaltung

7.4 Der Pförtner des Systems

Für die gesamte Modellierung sei vorausgesetzt, daß Prozesse zur Ausführung von Benutzeraufträgen erzeugt werden. Wir führen eine Zugangskontrolle zum System durch: Es werden nur die Aufträge von berechtigten Benutzern entgegengenommen. Prozesse werden ausschließlich zur Ausführung der Aufträge erzeugt, die diese Zugangskontrolle passieren. Die Komponente, die diese Kontrolle übernimmt, nennen wir *Pförtner*. Die Menge der berechtigten Benutzer sei beschrieben durch die Menge $UserId$.

Die Beschreibungen der Aufträge, die an das System gesendet werden, seien in einer abstrakten Auftragssprache $OrderL$ formuliert. Wir setzen voraus, daß anhand einer Auftragsbeschreibung $order \in OrderL$ alle Informationen bestimmt werden können, die benötigt werden, um einen Prozeß zu erzeugen: die Größe des benötigten Speicherplatzes, der Identifikator der Familie, der der Prozeß angehört, der Prozeßidentifikator, die Seiten, durch die das auszuführende Programm beschrieben wird, sowie das Programm in der abstrakten Maschinensprache $STEP_{Koop}$.

Ein Prozeß bildet gemäß der Modellierung in Abschnitt 4.3.3 die Schnittstelle zum Benutzer. Wir gehen davon aus, daß der Pförtner die allgemeine Schnittstelle für die Ein- und Ausgaben des modellierten Systems liefert. Daher stellt er die für jeden Prozeß benötigten Ports zur Umgebung bereit. Für den Pförtner sei eine ausreichende Anzahl privater Ports definiert. Die Menge der dem Pförtner initial zugeordneten privaten Kanäle wird festgelegt durch $N_{PFpp} = \{?!In_k, ?!Out_k \mid k \in I\!N\}$.

Kanal n	Nachrichtenmengen S_n
In_{PF}	$\{(o,u) \mid o \in OrderL, u \in UserId\}$
Out_{PF}	$\{LoginOk, LoginFailed\}$
$PFtoPQ$	$OrderL$
	$\cup\ ?!N_{BS}$ (für alle drei Kanäle)

Tabelle 7.4.1: Nachrichtentypen für die Komponente PF

Aus dem in Abbildung 7.3.1 gezeigten strukturellen Aufbau des Systems BS entnehmen wir die Schnittstelle des Pförtners und erhalten die in Abbildung 7.4.1 gezeigte ANDL-Spezifikation. Die Festlegung der Nachrichtentypen für die hier genannten Kanäle erfolgt mit Tabelle 7.4.1. Die den Kanälen In_k und Out_k zugeordneten Nachrichtentypen wurden bereits in Abschnitt 4.3.3 definiert.

agent PF
  input channels $In_{PF} : S_{Out_{PF}}$
  output channels $Out_{PF} : S_{Out_{PF}}, PFtoPQ : S_{PFtoPQ}$
  private channels $\ldots, ?!In_i : S_{In_i}, ?!Out_i : S_{Out_i}, \ldots$
is basic
  $\boldsymbol{f}_{\mathsf{PF}}$ mit der Spezifikation von Seite 165
end PF

Abbildung 7.4.1: ANDL-Spezifikation von PF

Der Pförtner erhält über seinen Eingabekanal In_{PF} Aufträge $order \in OrderL$ jeweils mit zugeordnetem Benutzeridentifikator. PF überprüft, ob der Benutzer über eine Zugangsberechtigung verfügt. Ist dies der Fall, wird *order* zum System zugelassen. Wird ein Benutzer ohne Berechtigung identifiziert, so wird der Auftrag abgewiesen. In der Spezifikation verwenden wir einen Zustandsparameter $i \in I\!N$, um die nächsten *freien* privaten Ports $?!In_i$ und $?!Out_i$ zu identifizieren. Wir beschreiben das Verhalten durch:

(1) PF wird durch den Erhalt eines ersten Auftrags $(order, login)$ mit $login \in UserId$ über Kanal In_{PF} gestartet, sendet die Nachrichten $LoginOk$ über Kanal Out_{PF} und $order$ mit den Ports $?!In_1$ und $?!Out_1$ über Kanal $PFtoPQ$. PF geht in den Zustand „2" über. $?!In_2$ bzw. $?!Out_2$ sind die als nächstes zu vergebenden Ports.

(2) Erhält PF im Zustand $k \in I\!N$ über Kanal In_{PF} eine Nachricht $(order, login)$ und gilt $login \in UserId$, werden die Nachrichten $LoginOk$ über Kanal Out_{PF} und $order$ sowie die Ports $?!In_k$ und $?!Out_k$ über Kanal $PFtoPQ$ gesendet. PF geht in den Zustand „$k+1$" über.

(3) Erhält PF im Zustand $k \in I\!N$ über Kanal In_{Pf} eine Nachricht $(order, login)$ und gilt $login \notin UserId$, wird die Nachricht $LoginFailed$ über Kanal Out_{PF} gesendet. PF verbleibt im Zustand k.

Wir erhalten die folgende formale Spezifikation:

Funktionsgleichungen für f_{PF}

$\forall s \in \prod_{n \in N_{BS}} [S_n^*], \quad order \in OrderL, \quad k \in I\!N \;:\; \exists h \in I\!N \times Type_{PF} \;:$

Für $login \in UserId$:

(1) $$f_{PF}(\{In_{PF} \mapsto \langle (order, login) \rangle\} \circ s) = \{Out_{PF} \mapsto \langle LoginOk \rangle, PFtoPQ \mapsto \langle order, ?!In_1, ?!Out_1 \rangle\} \circ h(2)(s)$$

Für $login \in UserId$:

(2) $$h(k)(\{IN_{PF} \mapsto \langle (order, login) \rangle\} \circ s) = \{Out_{PF} \mapsto \langle LoginOk \rangle, PFtoPQ \mapsto \langle order, ?!In_k, ?!Out_k \rangle\} \circ h(k+1)(s)$$

Für $login \notin UserId$:

(3) $$h(k)(\{In_{PF} \mapsto \langle (order, login) \rangle\} \circ s) = \{Out_{PF} \mapsto \langle LoginFailed \rangle\} \circ h(k)(s)$$

7.5 Die Queue des Pförtners

Die Queue PQ zwischen Pförtner und Prozeßverwalter speichert die Aufträge zur Prozeßerzeugung, die der Pförtner für das System zugelassen hat. Wir haben bereits mehrere Queues spezifiziert, so daß wir hier nur wenige zusätzliche Erklärungen geben müssen. Für PQ ist folgendes Verhalten wesentlich: Ein Auftrag, für dessen Ausführung aktuell kein Prozeß erzeugt werden kann, weil nicht genügend Seitenrahmen frei sind, darf nicht verloren gehen. Die zugehörige Beschreibung wird vom Prozeßverwalter an PQ zurückgegeben. PQ empfängt sowohl über Kanal $PFtoPQ$ als auch über Kanal $PVtoPQ$ Auftragsbeschreibungen, die abgespeichert werden. Ausgehend vom strukturellen Aufbau des erweiterten Systems in Abbildung 7.3.1 erhalten wir die in Abbildung 7.5.1 gezeigte ANDL-Spezifikation.

agent PQ
  input channels $PFtoPQ : S_{PFtoPQ}, PVtoPQ : S_{PVtoPQ}$
  output channels $PQtoPV : S_{PQtoPV}$
  private channels $\emptyset$
is basic
  $\boldsymbol{f_{PQ}}$ $\boxed{\text{mit der Spezifikation von Seite 167}}$
end PQ

Abbildung 7.5.1: ANDL-Spezifikation von PQ

Ausgehend von den in Abschnitt 7.4 angegebenen Festlegungen definieren wir die Nachrichtentypen gemäß Tabelle 7.5.1. Auf den Kanal $PFtoPQ$ verzichten wir, da er bereits in Tabelle 7.4.1 aufgeführt ist.

Kanal n	Nachrichtenmengen S_n
$PVtoPQ$	$\{Next\} \cup OrderL$ $\cup\ ?!N_{BS}$
$PQtoPV$	$\{Empty\} \cup OrderL$ $\cup\ ?!N_{BS}$

Tabelle 7.5.1: Nachrichtentypen für die Komponente PQ

PQ empfängt vom Pförtner neben den Auftragsbeschreibungen auch Ports zu Ein- und Ausgabekanälen, die an den Prozeßverwalter weitergegeben werden. Um die Zuordnung von Ports und Aufträgen zu dokumentieren, verfügt PQ über zwei interne Parameter. Mit $p \in OrderL^*$ beschreiben wir die Liste der gespeicherten Aufträge. Mit $q \in I\!N^*$ führen wir eine Liste, mit der die empfangenen Ports $?!In_k$ und $?!Out_k$ mit $k \in I\!N$ identifiziert werden. Das Verhalten der Queue wird textuell wie folgt beschrieben.

(1) Erhält PQ über Kanal $PFtoPQ$ eine erste Sequenz $\langle order, ?!In_1, ?!Out_1 \rangle$, so wird diese über Kanal $PQtoPV$ gesendet.

(2) Es gilt $p \neq \langle\rangle$. Erhält PQ über Kanal $PVtoPQ$ eine Anfrage $Next$ oder eine Anfrage $Next$ und einen Auftrag $order'$ mit zugehörigen Ports, werden die Nachricht $p[1]$ und die zugehörigen Ports $?!In_{q[1]}$ und $?!Out_{q[1]}$ über Kanal $PQtoPV$ gesendet, und $order'$ wird gespeichert. Alle über Kanal $PFtoPQ$ eintreffenden Aufträge $order$ werden ebenfalls gespeichert.

$p[1]$ identifiziert den gemäß der internen Liste nächsten Auftrag und $q[1]$ die zugehörigen Ports $?!In$ und $?!Out$.

(3) Es gilt $p = \langle\rangle$, und PQ erhält über Kanal $PFtoPQ$ aktuell keinen Auftrag.

  (a) Erhält PQ über Kanal $PVtoPQ$ die Anfrage $Next$, wird die Nachricht $Empty$ über Kanal $PQtoPV$ gesendet.

(b) Erhält PQ über Kanal $PVtoPQ$ zusätzlich eine Nachricht $order$ mit zugehörigen Ports, wird diese erneut über Kanal $PQtoPV$ gesendet.

(4) Es gilt $p = \langle\rangle$. Erhält PQ über Kanal $PFtoPQ$ einen Auftrag $order$ mit zugehörigen Ports und über Kanal $PVtoPQ$ die Anfrage $Next$ oder die Anfrage $Next$ und einen Auftrag $order'$ mit zugehörigen Ports, sendet PF über Kanal $PQtoPV$ die Nachricht $order$ mit zugehörigen Ports und speichert $order'$ ab.

Diese textuelle Beschreibung wird umgesetzt in die formale Spezifikation:

Funktionsgleichungen für f_{PQ}

$$\forall s \in \prod_{n \in N_{BS}} [S_n^*], \quad order, order' \in OrderL, \; k, l \in I\!N, \quad p, p_{new} \in OrderL^*, q, q_{new} \in I\!N^*,$$
$$v \in \{\langle\rangle, \langle order, ?!In_l, ?!Out_l\rangle\}, \quad w \in \{\langle Next\rangle, \langle Next, order', ?!In_l, ?!Out_l\rangle\} \;:$$
$$\exists h \;\in\; (OrderL^* \times I\!N^*) \rightarrow Type_{PQ} \;:$$

(1) $f_{PQ}(\{PFtoPQ \mapsto \langle order, ?!In_1, ?!Out_1\rangle\} \circ s)$
$= \{PQtoPV \mapsto \langle order, ?!In_1, ?!Out_1\rangle\} \circ h(\langle\rangle, \langle\rangle)(s)$

Für $p \neq \langle\rangle$:

(2) $h(p, q)(\{PFtoPQ \mapsto v, PVtoPQ \mapsto w\} \circ s)$
$= \{PQtoPV \mapsto \langle p[1], ?!In_{q[1]}, ?!Out_{q[1]}\rangle\} \circ h(rt.p \circ p_{new}, rt.q \circ q_{new})(s)$

wobei $p_{new} = OrderL Ⓒ (v \circ w)$
$(\#k Ⓒ q_{new} = 1 \Leftrightarrow \#?!In_k Ⓒ (v \circ w) = 1) \;\;\wedge\;\; (\#k Ⓒ q_{new} = 0 \Leftrightarrow \#?!In_k Ⓒ (v \circ w) = 0)$

(3a) $h(\langle\rangle, \langle\rangle)(\{PFtoPQ \mapsto \langle\rangle, PVtoPQ \mapsto \langle Next\rangle\} \circ s) = \{PQtoPV \mapsto \langle Empty\rangle\} \circ h(\langle\rangle, \langle\rangle)(s)$

(3b) $h(\langle\rangle, \langle\rangle)(\{PFtoPQ \mapsto \langle\rangle, PVtoPQ \mapsto \langle Next, order, ?!In_l, ?!Out_l\rangle\} \circ s)$
$= \{PQtoPV \mapsto \langle order, ?!In_l, ?!Out_l\rangle\} \circ h(\langle\rangle, \langle\rangle)(s)$

(4) $h(\langle\rangle, \langle\rangle)(\{PFtoPQ \mapsto \langle order, ?!In_k, ?!Out_k\rangle PVtoPQ \mapsto w\} \circ s)$
$= \{PQtoPV \mapsto \langle order, ?!In_k, ?!Out_k\rangle\} \circ h(p_{new}, q_{new})(s)$

wobei $p_{new} = OrderL Ⓒ w \;\wedge$
$(\#k Ⓒ q_{new} = 1 \Leftrightarrow \#?!In_k Ⓒ w = 1) \;\;\wedge\;\; (\#k Ⓒ q_{new} = 0 \Leftrightarrow \#?!In_k Ⓒ w = 0)$

7.6 Der Prozeßverwalter

Der Prozeßverwalter PV ist dafür zuständig, Prozesse zur Ausführung der Aufträge zu erzeugen, die der Pförtner für das System zugelassen hat. PV bildet die *Schnittstelle* zu allen weiteren Verwaltungseinheiten des von uns modellierten Systems. Insgesamt rundet er die Modellierung eines in FOCUS modellierten *Betriebssystems* ab. Für die benötigte

Einbindung des Prozeßverwalters in das bestehende System orientieren wir uns daran, welche Schritte ausgeführt werden, um einen neuen Prozeß in das bestehende System zum Ressourcenmanagement zu integrieren. Diese umfassen:

1. zunächst und vor allem anderen die Anbindung an das Subsystem der Speicherverwaltung. Hier wird eine Verbindung zum globalen Speicherverwalter GS benötigt, da GS über die Vergabe von Speicherplatz entscheidet. Erst mit dem „Ok“ des Speicherverwalters kann der neue Prozeß erzeugt werden.

2. zur Integration des neuen Prozesses in die Prozessorverwaltung eine Verbindung zur dem Dispatcher zugeordneten Queue und damit dem Scheduler.

3. die Verbindung zu den Kooperationsverwaltern aller Prozeßfamilien, um den neuen Prozeß in *seine* Familie einzubinden.

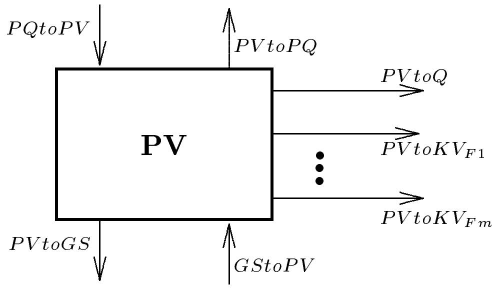


Abbildung 7.6.1: SSD des Prozeßverwalters

Wir erhalten die in Abbildung 7.6.1 gezeigte Schnittstelle des Prozeßverwalters. Im System `BS` sind die Familien $F1$ bis Fm und deren Kooperationsverwalter enthalten. Über die dabei verwendeten Kanäle $PVtoKV_{Fj}$ mit $j \in \{1, \ldots, m\}$ und $PVtoQ$ werden ausschließlich Ports verschickt, die zugeordnete Nachrichtenmenge entspricht somit der Menge $?!N_{BS}$. Die Kanälen $PQtoPV$ und $PVtoPQ$ sind bereits in Tabelle 7.5.1 aufgeführt, und die Nachrichtenmengen der Kanäle $PVtoGS$ und $GStoPV$ werden in Abschnitt 7.7.2 erklärt.

Für das System `BS` sei $Pid = \{Pi \mid i \in I\!N\}$ die Menge der Prozeßidentifikatoren. PV verfügt initial über die Menge privater Kanäle, die für die Einbindung des neuen Prozesses in das System benötigt werden. Die den privaten Kanälen zugeordneten Nachrichtenmengen können jeweils in den vorangegangenen Kapiteln nachgelesen werden. Die privaten Kanäle von PV umfassen die in Abbildung 7.6.2 genannten Mengen:

- `Disp` zur Integration des Prozesses in das Subsystem der Prozessorverwaltung und dabei zur Anbindung des Prozesses an die Queue des Dispatchers und den Dispatcher. Es gilt:

$$\texttt{Disp} \;=\; \{?!PitoQ, ?!DtoPi \mid Pi \in Pid\}$$

- `FamFj` zur Integration eines Prozesses in die Familie Fj mit $j \in \{1, \ldots, m\}$. Zu dieser Menge gehören die Ein- und Ausgabeports zur Verbindung mit dem Kooperationsverwalter KV_{Fj} der Familie. Es gilt:

$$\texttt{FamFj} \;=\; \{?!PitoKV_{Fj}, ?!KV_{Fj}toPi \mid Pi \text{ gehört zu Familie } Fj\}$$

- `Koop` zur Realisierung des Nachrichtenaustauschs. Wird das System mit nichtblockierendem Senden modelliert, gilt `Koop` $= \emptyset$; bei blockierendem Senden werden einem Prozeß die privaten Kanäle zur Durchführung des Rendezvous mitgegeben. Es gilt:

$$\texttt{Koop} \;=\; \{?!ConftoPi \mid Pi \in Pid\}$$

Mit den oben gegebenen Erklärungen und dem in Abbildung 7.6.1 gezeigten SSD erhalten wir die in Abbildung 7.6.2 gezeigte ANDL-Spezifikation von PV.

agent PV
  input channels $PQtoPV \,:\, S_{PQtoPV}, GStoPV \,:\, S_{GStoPV}$
  output channels $PVtoPQ \,:\, S_{PVtoPQ}, PVtoGS \,:\, S_{PVtoGS}, PVtoQ \,:\, S_{PVtoQ},$
  $PVtoKV_{F1} \,:\, S_{PVtoKV_{F1}}, \ldots, PVtoKV_{Fm} \,:\, S_{PVtoKV_{Fm}}$
  private channels `Disp`, `FamF1`, ..., `FamFm`, `Koop`
is basic
  f_{PV} $\boxed{\text{mit der Spezifikation von Seite 171}}$
end PV

Abbildung 7.6.2: ANDL-Spezifikation von `PV`

Zur Bestimmung der Informationen, die zur Prozeßerzeugung benötigt und in unserer Modellierung aus einem Auftrag $order \in OrderL$ hergeleitet werden, definieren wir die folgenden Hilfsfunktionen.

Die Funktion $Prog$ bestimmt aus $order$ das auszuführende Programm in der von uns verwendeten abstrakten Maschinensprache $STEP_{Koop}$. Jeder Instruktion ist eine virtuelle Adresse zugeordnet. Mit den Bezeichnungen aus den Kapiteln 4, 5 und 6 ist $STEP_{Koop}$ die Menge der Anweisungen in Maschinensprache und VA_{Pi} die Menge der virtuellen Adressen eines Prozesses. Mit $Prog(order)$ erhalten wir eine Sequenz von Paaren jeweils bestehend aus einem Berechnungsschritt $Step_i$ und der zugehörigen virtuellen Adresse va_i. Die Funktion $Prog$ ist somit definiert durch:

$$Prog \;:\; OrderL \;\longrightarrow\; (STEP_{Koop} \times \bigcup_{pid \in Pid} VA_{pid})^*$$

Der Prozeßidentifikator wird durch die Funktion Id und der Identifikator der Familie, in die der Prozeß eingeordnet wird, durch die Funktion Fam bestimmt. Es gelten

$$Id \;:\; OrderL \;\longrightarrow\; Pid \qquad \text{und} \qquad Fam \;:\; OrderL \;\longrightarrow\; \{F1, \ldots, Fm\}$$

Ein Prozeß kann nur erzeugt werden, wenn freier Speicherplatz in ausreichender Größe vorhanden ist. Wir gehen davon aus, daß jedem Prozeß eine feste Anzahl von Seitenrahmen im Arbeitsspeicher zur exklusiven Nutzung zur Verfügung steht, siehe Kapitel 5. Der globale Speicherverwalter benötigt Informationen über die Anzahl von Seitenrahmen und die Seiten, durch die ein Prozeß beschrieben wird. Dies wird durch die Funktion Mem bestimmt, die wir in Abschnitt 7.7.2 erklären.

Der *Prozeßverwalter* entnimmt aus PQ einen nächsten Auftrag $order$ und überprüft mit Hilfe des globalen Speicherverwalters, ob genügend freie Seitenrahmen zur Verfügung stehen. Ist dies der Fall, wird ein Prozeß $Pj = Id(order)$ mit dem entsprechenden Programm erzeugt, sobald der Speicherplatz belegt ist. Pj wird der private Speicherverwalter SV_{Pj} zugeordnet, und beide werden in das bestehende System integriert. Dabei wird Pj initialisiert mit dem Programm $Prog(order)$ und den Verbindungen zum Dispatcher, dem Kooperationsmanager $KV_{Fam(order)}$ und SV_{Pj}. Ist die Erzeugung eines Pj abgeschlossen, hat der Prozeßverwalter keine direkte Verbindung mehr zu Pj. Für das Verhalten gilt:

(1) Erhält PV die Nachricht $order$ und Ports $?!In_l$ sowie $?!Out_l$ über Kanal $PQtoPV$, sendet er die Sequenz $\langle Mem(order), Id(order) \rangle$ über Kanal $PVtoGS$ und speichert $order$ und l.

PV wartet auf die Rückmeldung des globalen Speicherverwalters und speichert den Auftrag sowie den Index der Ports $?!In$ und $?!Out$.

(2) Erhält PV über Kanal $PQtoPV$ die Nachricht $Empty$, sendet er die Nachricht $Next$ über Kanal $PVtoPQ$.

(3) Erhält PV im Zustand $(order, l)$ die Nachricht $NoMem$ über Kanal $GStoPV$, sendet er über Kanal $PVtoPQ$ die Sequenz $\langle Next, order, ?!In_l, ?!Out_l \rangle$.

Der Prozeß kann nicht erzeugt werden, da nicht genügend Seitenrahmen frei sind.

(4) Es gelten $Pj = Id(order)$ und $Fk = Fam(order)$.
Erhält PV im Zustand $order$ über Kanal $GStoPV$ die Nachricht $MemOk$ sowie die Ports $?SVtoPj$ und $!PjtoSV$, wird ein neuer Prozeß Pj mit passender initialer Schnittstelle erzeugt. Zudem sendet PV die Nachricht $Next$ über Kanal $PVtoPQ$.

Ein Prozeß $Id(order) = Pj$ wird gemäß seiner Einordnung in das System mit einer initialen Schnittstelle versehen. Diese umfaßt die Kanäle zur Umgebung, zur Queue des Dispatchers und zum Kooperationsverwalter der Familie. Zusätzlich erhält Pj die privaten Ports, um sich beim Dispatcher anmelden zu können, und, in einem System mit nichtblockierendem Senden, die privaten Kanäle, um das Rendezvous durchzuführen. Gleichung (4) ergibt sich mit diesen Erläuterungen gemäß Schema (2.19) zur Erzeugung einer neuen Komponente aus Abschnitt 2.6. In (4) senden wir das Programm $Prog(order) \in (STEP \times VA_{Pi})^*$ über Kanal In_i an den Prozeß. Der Vergleich mit der Spezifikation eines Prozesses in Abschnitt 4.3.3 zeigt, daß ein Prozeß auf diese Weise gestartet wird. Wir verzichten auf die explizite Aufnahme des Zustands *started* in die Menge $State_{Pi}$, da dieser Fall in unserer Modellierung implizit spezifiziert wird.

Funktionsgleichungen für f_{PV}
$\forall s \in \prod_{n \in N_{BS}} [S_n^*], \quad order \in OrderL, l \in I\!N \;:\; \exists h \in (OrderL \times I\!N) \to Type_{PZ}, \quad f_{Pj} \in [\![Pi]\!] \;:$
(1) $f_{PV}(\{PQtoPV \mapsto \langle order, ?!In_l, ?!Out_l\rangle\} \circ s) = \{PVtoGS \mapsto \langle Mem(order), Id(order)\rangle\} \circ h(order,l)(s)$
(2) $f_{PV}(\{PQtoPV \mapsto \langle Empty\rangle\} \circ s) = \{PVtoPQ \mapsto \langle Next\rangle\} \circ f_{PV}(s)$
(3) $h(order,l)(\{GStoPV \mapsto \langle NoMem\rangle\} \circ s) = \{PVtoPQ \mapsto \langle Next, order, ?!In_l, ?!Out_l\rangle\} \circ f_{PV}(s)$
Mit $Pj \triangleq Pid(order)$ sowie $Fk \triangleq Fam(order)$:
(4) $h(order,l)(\{GStoPV \mapsto \langle MemOk, ?SVtoPj, !PjtoSV\rangle\} \circ s) = \{PVtoPQ \mapsto \langle Next\rangle, PVtoQ \mapsto \langle ?PjtoQ\rangle, PVtoKV_{Fk} \mapsto \langle !KV_{Fk}toPj, ?PjtoKV_{Fk}\rangle\} \circ (f_{PV} \otimes f_{Pj}(\mathtt{PjInit}))(\{Out_l \mapsto \langle !In_l, ?Out_l\rangle, In_l \mapsto \langle Prog(order)\rangle\} \circ s)$
wobei $\mathtt{PjInit} = \{?In_l, !Out_l, ?SVtoPj, !PjtoSV, ?KV_{Fk}toPj, !PjtoKV_{Fk}, !PjtoQ, ?!DtoPj\}$ **ohne** Rendezvous
und $\mathtt{PjInit} = \{?In_l, !Out_l, ?SVtoPj, !PjtoSV, ?KV_{Fk}toPj, !PjtoKV_{Fk}, !PjtoQ, ?!DtoPj, \boldsymbol{?!ConftoPj}\}$ **mit** Rendezvous

7.7 Erweiterung des Speicherverwalters

Im folgenden erweitern wir die Modellierung des globalen Speicherverwalters GS aus Abschnitt 5.6 um die Erzeugung eines Prozesses und dessen initiale Speicherbelegung. Diese Erweiterung ist dadurch charakterisiert, daß lokale Speicherverwalter SV_{Pi} bei Bedarf von GS erzeugt werden. Ein neu erzeugter SV_{Pi} wird direkt mit den prozeßlokalen Informationen initialisiert. Diese umfassen eine Liste der dem Prozeß exklusiv zugeordneten Seitenrahmen und die Blöcke, durch die der Prozeß initial im Hintergrundspeicher realisiert ist. Wir vermeiden durch diese Vorgehensweise, daß GS die prozeßlokalen Informationen bzgl. der Speicherverwaltung an den Prozeßverwalter senden muß.

Die Modellierung von GS wird um eine dritte interne Komponente SE erweitert, die lokale Speicherverwalter erzeugt und die initiale Realisierung der Seiten des Prozesses im Hintergrundspeicher vornimmt. SE verfügt über jeweils eine Liste freier Seitenrahmen und Blöcke für den gesamten Arbeits- bzw. Hintergrundspeicher. SE erhält von der Queue SQ des Speicherverwalters die Identifikatoren frei gewordener Seitenrahmen und Blöcke. Mit der Komponente SW ist SE durch einen Kanal, auf den er schreibend zugreift, verbunden und bindet einen neuen lokalen Speicherverwalter auf diesem Weg an GS. Der neue Aufbau von GS ist durch das in Abbildung 7.7.1 gezeigte SSD dargestellt. Mit der ANDL-Spezifikation wird die initiale Vernetzung des verteilten Systems angegeben. Diese ergibt sich mit dem oben gezeigten SSD und der initialen Struktur des Systems, in der kein Prozeß enthalten ist. Wir verzichten hier auf die explizite Darstellung dieser ANDL-Spezifikation, die sich aus der in Abschnitt 5.6 mit Abbildung 5.6.2 gezeigten Spezifikation ergibt.

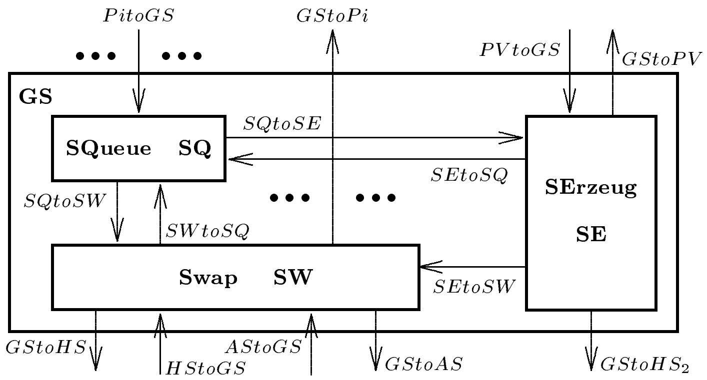


Abbildung 7.7.1: Der erweiterte globale Speicherverwalter GS

Da die Erweiterung der Komponente SW nur die Hinzunahme eines weiteren Kanals zur Schnittstelle umfaßt, über den ausschließlich Ports gesendet werden, verzichten wir auf die Formalisierung. Im folgenden zeigen wir die Erweiterung der Komponente SQ und die Spezifikation der neuen Komponente SE.

7.7.1 Erweiterung der Queue des Speicherverwalters

Die Speicherqueue SQ bildet, siehe Abschnitt 5.6.1, die Schnittstelle zu den prozeßlokalen Speicherverwaltern SV_{Pi} und nimmt von diesen die Aufträge zum Umspeichern von Seiten entgegen. Da wir in diesem Abschnitt die Modellierung des Systems dadurch abrunden, daß lokale Speicherverwalter erzeugt und aufgelöst werden, ist eine Erweiterung der Modellierung von SQ notwendig. Entsprechend zu allen bisher gezeigten Spezifikationen sind die Veränderungen in dieser Modellierung **markiert**.

Bei der Erzeugung eines lokalen Speicherverwalters wird dieser SQ bekannt gegeben. Mit der Auflösung eines Prozesses Pi und damit von SV_{Pi} wird Speicherplatz frei. Die dadurch wieder zur Verfügung stehenden Seitenrahmen und Blöcke werden als *freier* Speicherplatz registriert. Für die Buchführung über die freien Seitenrahmen und Blöcke ist die Komponente SE zuständig. SQ wird so erweitert, daß alle Identifikatoren frei gewordener Seitenrahmen und Blöcke von SQ an SE weitergeleitet werden. Wir erhalten die in Abbildung 7.7.1 gezeigte erweiterte ANDL-Spezifikation, wobei wir, im Gegensatz zu Abbildung 5.6.3 in Abschnitt 5.6.1, die initiale Vernetzung der Speicherqueue angeben, in der weder Prozesse noch lokale Speicherverwalter enthalten sind.

Die Nachrichtentypen der mit der Modellierung in Abschnitt 5.6.1 bereits bekannten Kanäle können auf der Seite 125 nachgelesen werden. SQ sendet an SE Seitenrahmen- und Block-

agent SQ
input channels $SWtoSQ \ : \ S_{SWtoSQ}, \boldsymbol{SEtoSQ} \ : \ \boldsymbol{S_{SEtoSQ}}$
output channels $SQtoSW \ : \ S_{SQtoSW}, \boldsymbol{SQtoSE} \ : \ \boldsymbol{S_{SQtoSE}}$
private channels $\emptyset$
is basic
f_{SQ} | mit der Spezifikation von Seite 126 und den Erweiterungen von Seite 173 |
end SQ

Abbildung 7.7.1: ANDL-Spezifikation von SQ

identifikatoren der in einem Zeitintervall frei gewordenen Speicherplätze. Für den neu hinzugenommenen Kanal $SQtoSE$ ist somit die Nachrichtenmenge $S_{SQtoSE} = SRid \cup Bid \cup N_{BS}$ definiert. Über den Kanal $SEtoSQ$ werden ausschließlich Ports verschickt.

Die Erweiterung des Verhaltens von SQ umfaßt die Registrierung frei gewordener Seitenrahmen und Blöcke. Die Modellierung der Terminierung eines lokalen SV_{Pi} in Abschnitt 7.8.2 wird zeigen, daß SQ immer bereit sein muß, Informationen über frei gewordenen Speicherplatz an SE weiterzuleiten. Die textuelle Beschreibung von SQ aus Abschnitt 5.6.1 wird in jedem der Punkte (1), (2) und (3) von Seite 125 um folgenden Zusatz erweitert:

> Alle über $\boldsymbol{PitoSV}$ mit $PitoSV \in ap_{SQ}$ empfangenen Seitenrahmen- und Blockidentifikatoren werden über $\boldsymbol{SQtoSE}$ weitergeleitet.

Jede Gleichung der in Abschnitt 5.6.1 gezeigten Spezifikation wird gemäß diesem textuellen Zusatz erweitert. Zudem muß berücksichtigt werden, daß sich im System BS die Anzahl der Prozesse verändert. Alle aktiven Prozesse sind mit GS jeweils über einen Ein- und einen Ausgabekanal verbunden. Eine Überprüfung der aktiven Schnittstelle ap_{SQ} liefert die im System vorhandenen Prozesse. Wir zeigen beispielhaft die Umsetzung von Punkt (3a). Dabei verwenden wir die in Abschnitt 5.6.1 eingeführte Menge $SwapMsg$.

Erweiterung der Spezifikation für f_{SQ} von Seite 126

$$\forall s \in \prod_{n \in N_{BS}} [S_n^*],\ k, l, i_l \in I\!N,\ P(i_l)toSV \in ap_{SQ},$$
$$in_k \in SwapMsg \ \cup \ \{\langle u, v, ?SVto(pid), !(pid)toSV \rangle \mid u \in SRid^*, v \in Bid^*, pid \in Pid\} \ :$$
$$\exists\ h \in SwapMsg^* \ \rightarrow \ Type_{SQ} \ :$$

Es gibt ein $i_l \in \{i_1, \ldots i_n\}$ mit $SwapMsg \copyright in_{i_l} \neq \langle\rangle$:

$$(3a) \quad h(\langle\rangle)(\{P(i_1)toSV \mapsto \langle in_{i_1} \rangle, \ldots, P(i_n)toSV \mapsto \langle in_{i_n} \rangle, SWtoSQ \mapsto \langle Next \rangle\} \circ s)$$
$$= \{SQtoSW \mapsto \langle ft.\mathtt{Swap} \rangle, \boldsymbol{SQtoSE} \mapsto \langle \mathtt{Frame}, \mathtt{Block} \rangle\} \circ h(rt.\mathtt{Swap})(s)$$

wobei $\mathtt{Swap} = SwapMsg \copyright (in_{i_1} \circ \ldots \circ in_{i_n})$,
$\mathtt{Frame} = (frame, SRid \copyright in_{i_1} \circ \ldots \circ in_{i_n})$ und $\mathtt{Block} = (block, Bid \copyright in_{i_1} \circ \ldots \circ in_{i_n})$
$(k \in \{i_1, \ldots, i_n\} \ \Leftrightarrow \ PktoSV \in ap_{SQ}) \ \wedge \ (k \notin \{i_1, \ldots, i_n\} \ \Leftrightarrow \ PktoSV \notin ap_{SQ})$

7.7.2 Speicherverwaltung zur Prozeßerzeugung

GS überprüft die Voraussetzungen für die Erzeugung eines Prozesses in Bezug auf die Speicherverwaltung und realisiert den Prozeß, falls diese Voraussetzungen erfüllt sind. Die Realisierung umfaßt die Zuteilung freier Seitenrahmen und die Zuweisung der Seiten, die den Prozeß beschreiben, an Blöcke des Hintergrundspeichers. Im System `BS` übernimmt die neu hinzugenommene Komponente SE diese Aufgaben. Mit SQ ist SE durch zwei Kanäle verbunden. Über den Kanal $SQtoSE$, auf den sie lesend zugreift, empfängt sie Listen frei gewordener Seitenrahmen und Blöcke. Über den Kanal $SEtoSQ$, den sie schreibend nutzt, sendet SE ausschließlich Ports. Mit der Komponente SW ist SE durch den Kanal $SEtoSW$ verbunden, wobei sie über das Schreibrecht verfügt. Auch über diesen Kanal werden ausschließlich Ports verschickt. Bei erfolgreicher Erzeugung eines lokalen Speicherverwalters SV_{Pi} werden das Leserecht an Kanal $PitoGS$ über Kanal $SEtoSQ$ und das Schreibrecht an Kanal $GStoPi$ über Kanal $SEtoSW$ verschickt. Auf diesem Weg wird die erzeugte Komponente SV_{Pi} an den globalen Speicherverwalter gebunden und in die Speicherverwaltung integriert.

Ausgehend von dem SSD für GS in Abbildung 7.7.1 zeigt Abbildung 7.7.2 die ANDL-Spezifikation für SE. SE wird mit allen für die Erzeugung der lokalen Speicherverwalter benötigten privaten Ports initialisiert.

agent SE
  input channels $PVtoGS : S_{PVtoGS}, SQtoSE : S_{SQtoSE}$
  output channels $GStoPV : S_{GStoPV}, SEtoSQ : S_{SEtoSQ},$
  $SEtoSW : S_{SEtoSW}, GStoHS_2 : S_{GStoHS_2}$
  private channels `GStoSV`, `SVtoProc`, `AtoLRU`
is basic
  $\boldsymbol{f_{SE}}$ mit der Spezifikation von Seite 177
end SE

Abbildung 7.7.2: ANDL-Spezifikation von SE

Entsprechend zur ANDL-Spezifikation des Prozeßverwalters in Abbildung 7.6.2 wurden die Mengen der privaten Kanäle von SE abgekürzt:

- `GStoSV` ist die Menge der Ports für die Verbindungen aller lokalen Speicherverwalter zum globalen Speicherverwalter. Sie ist definiert durch

$$\texttt{GStoSV} = \{?!GStoPi, ?!PitoGS \mid Pi \in Pid\}$$

- `SVtoProc` ist die Menge der Ports für die Verbindungen jedes lokalen Speicherverwalters zu *seinem* Prozeß. Es gilt:

$$\texttt{SVtoProc} = \{?!SVtoPi, ?!PitoSV \mid Pi \in Pid\}$$

- `AtoLRU` enthält alle Ports zur internen Kopplung der Komponenten A_{Pi} mit der zugehörigen Komponente LRU_{Pi}. Es gilt:

$$\texttt{AtoLRU} \;=\; \{?!LtoPi, ?!PitoL \mid \;Pi \in Pid\}$$

Für die Kanäle $GStoPV$ und $PVtoGS$ sind die in Tabelle 7.7.1 gezeigten Nachrichtentypen festgelegt. Die den privaten Ports zugeordneten Nachrichtenmengen können in Kapitel 5 nachgelesen werden.

Kanal n	Nachrichtenmengen S_n
$PVtoGS$	$Pages^* \times Pid$
$GStoPV$	$\{MemOk, NoMem\}$
	$\cup \; ?!N_{BS}$ (für beide Kanäle)

Tabelle 7.7.1: Nachrichtentypen für die Komponente SE

Mit der in Abschnitt 7.6 bereits genannten Funktion Mem wird aus $order \in OrderL$ die Information bestimmt, die für die Belegung von Speicherplatz und die Realisierung des Prozesses im Hintergrundspeicher benötigt wird. Mem bestimmt eine Liste von Seiten des Prozesses $slist \in Pages^*$. Wir setzen voraus, daß $slist$ so geordnet ist, daß $slist[1]$ die erste Seite, $slist[2]$ die zweite Seite, und $slist[\#slist]$ die letzte Seite des Prozesses darstellt. Die Zuordnung der Seiten zu ihren Identifikatoren ergibt sich gemäß dieser Liste. Die Länge der Liste bestimmt die Anzahl der benötigten Seitenrahmen. Diese Funktion wird definiert durch $Mem \;:\; OrderL \longrightarrow Pages^*$.

Als weitere Abkürzungen verwenden wir $SRInit$ und $BInit$ für die Liste aller Identifikatoren der Seitenrahmen des Arbeitsspeichers bzw. aller Blöcke des Hintergrundspeichers. Wir setzen voraus, daß es im Hintergrundspeicher immer ausreichend viele freie Blöcke gibt, und daß der erste Prozeß in einem Systemzustand erzeugt wird, in dem alle Seitenrahmen und Blöcke der Speicher frei sind.

Mit diesen Erklärungen zur Erzeugung eines lokalen Speicherverwalters geben wir die folgende textuelle Beschreibung des Verhaltens von SE. Mit $sr \in SRid^*$ und $b \in Bid^*$ bezeichnen wir jeweils die Liste der aktuell freien Seitenrahmen bzw. Blöcke.

(1) SE wird durch den Empfang der Nachricht $(slist, pid)$ gestartet. SE belegt die ersten $\#slist$ Seitenrahmen der Liste $SRInit$ und realisiert die Seiten in den ersten $\#slist$ Blöcken gemäß $BInit$. Der lokale Speicherverwalter SV_{pid} wird erzeugt und mit den erforderlichen Informationen initialisiert. Über Kanal $GStoPV$ werden die Nachricht $MemOk$ und die Ports $?SVto(pid)$ und $!(pid)toSV$ gesendet.

In der initialen Phase von SE kann der erste Prozeß pid direkt erzeugt werden, da zu Beginn eine ausreichende Anzahl freier Seitenrahmen zur Verfügung steht. SV_{pid} besteht aus den Komponenten A_{pid} und LRU_{pid}. Ein neu erzeugter SV_{pid} wird mit folgenden Informationen initialisiert: die Liste der Seitenrahmen, die dem Prozeß zu seiner exklusiven Nutzung zugeteilt sind, und die Zuordnung der Seiten zu den Blöcken, in denen sie initial realisiert wurden. Mit den Ports $?SVto(pid)$ und $!(pid)toSV$ werden die Zugriffsrechte an den Kanälen weitergegeben, die pid mit seinem SV_{pid} verbinden. Da pid erst nach erfolgreicher Realisierung im Speicher von PV erzeugt wird, benötigt PV diese Ports, um sie zur initialen Schnittstelle von pid hinzuzufügen.

(2) SE erhält über Kanal $PVtoGS$ die Nachricht $(slist, pid)$ und über Kanal $SQtoSE$ die Listen $freesr \in SRid^*$ und $freeb \in Bid^*$. Falls $\#slist > (sr \circ freesr)$ gilt, sendet SE die Nachricht $NoMem$ über Kanal $GStoPV$, und alle über Kanal $SQtoSE$ empfangenen Seiten- und Blockidentifikatoren werden in die Listen sr bzw. b aufgenommen.

Zur Erzeugung des Prozesse pid wird eine größere Anzahl von Seitenrahmen benötigt, als aktuell verfügbar sind. Die Erzeugung von pid ist nicht möglich.

(3) SE erhält über Kanal $PVtoGS$ die Nachricht $(slist, pid)$, über Kanal $SQtoSE$ die Listen $freesr \in SRid^*$ und $freeb \in Bid^*$, und es gilt $\#slist \leq (sr \circ freesr)$. SE belegt die ersten $\#slist$ Seitenrahmen der Liste $sr \circ freesr$ und realisiert die Seiten in den ersten $\#slist$ Blöcken gemäß $b \circ freeb$. Der lokale Speicherverwalter SV_{pid} wird erzeugt und mit den erforderlichen Informationen initialisiert. Über Kanal $GStoPV$ werden die Nachricht $MemOk$ und die Ports $?SVto(pid)$ und $!(pid)toSV$ gesendet. Alle über Kanal $SQtoSE$ empfangenen Seiten- und Blockidentifikatoren werden in die Listen sr bzw. b aufgenommen.

Zur Erzeugung von pid stehen genügend freie Seitenrahmen zur Verfügung und SV_{Pi} wird erzeugt. Siehe auch die Erklärungen zu Punkt (1).

Immer wenn Speicherplatz in ausreichender Größe verfügbar ist, werden mit der Erzeugung des Prozesses Blöcke belegt. Wir beschreiben dies abstrakt durch die Nachricht $Alloc$, die die Zuordnung von Blockidentifikatoren zu den Seiten enthält, die den Prozeß gemäß $slist$ beschreiben. Zur Initialisierung eines SV_{pid} wird die Information bzgl. der Zuordnung von Seiten zu Blöcken, die für die initiale Belegung des Arrays benötigt wird, über Kanal $GStoPi$ gesendet. Gemäß der vorgenommenen Realisierung der Seiten benennt $(q \circ freeb)[1]$ den Blockidentifikator, in dem die erste Seite, und $(q \circ freeb)[k]$ den Blockidentifikator, in dem die letzte Seite des Prozesses realisiert ist. Die weiteren Seiten sind entsprechend der Liste der Blockidentifikatoren realisiert.

Die in der folgenden Spezifikation verwendeten Abkürzungen `chA` und `chLRU` stehen für die Ports, die den Komponenten A_{pid} bzw. LRU_{pid} initial mitgegeben werden. Sie legen die Schnittstellen eines lokalen Speicherverwalters und seiner Unterkomponenten fest. Die Schnittstellen dieser Komponenten können in der Abbildung 5.5.1 von Abschnitt 5.5 der Modellierung der Speicherverwaltung abgelesen werden.

Funktionsgleichungen für f_{SE}

$\forall s \in \prod_{n \in N_{BS}} [S_n^*],\ k \in I\!N,\ pid \in Pid,\ slist \in Pages^*$
$\quad p, freesr, SRInit \in SRid^*,\ q, freeb, BInit \in Bid^*\ :$
$\exists\ h \in (SRid^* \times Bid^*) \ \to\ Type_{SE},\ f_{A_{pid}}\ \in\ [\![A_{Pi}]\!],\ f_{LRU_{pid}}\ \in\ [\![LRU_{Pi}]\!]\ \ :$

$$(1)\quad f_{SE}(\{PVtoGS \mapsto \langle\,(slist, pid)\,\rangle\} \circ s)$$
$$= \{GStoPV \mapsto \langle\, MemOk, ?SVto(pid), !(pid)toSV\,\rangle, GStoHS_2 \mapsto \langle \texttt{HSInit}\rangle\}$$
$$\circ\,(h(rt^k.SRInit, rt^k.BInit) \otimes (f_{A_{pid}}(\texttt{chA}) \otimes f_{LRU_{pid}}(\texttt{chLRU})))$$
$$(\{GSto(pid) \mapsto \langle \texttt{SVInit}\rangle, (pid)toL \mapsto \langle \texttt{LInit}\} \circ s)$$

wobei $\texttt{HSInit} = Alloc[\ (Binit[1], slist[1]),\ \ \ldots, (Binit[k], slist[k])\ ]$
$\texttt{LInit} = SRInit[1] \circ \ldots \circ SRInit[k],\ \ \texttt{SVInit} \triangleq (k, Binit[1] \circ \ldots \circ Binit[k])$

Für $k\ >\ p \circ freesr\ :$

$$(2)\quad h(p,q)(\{PVtoGS \mapsto \langle\,(slist, pid)\,\rangle, SQtoSE \mapsto \langle\,(frame, freesr), (block, freeb)\,\rangle\} \circ s)$$
$$= \{GStoPV \mapsto \langle NoMem\,\rangle\} \circ h(p \circ freesr, q \circ freeb)(s)$$

Für $k\ \leq\ p \circ freesr\ :$

$$(3)\quad h(p,q)(\{PVtoSV \mapsto \langle\,(slist, pid)\,\rangle, SQtoSE \mapsto \langle\,(frame, freesr), (block, freeb)\,\rangle\} \circ s)$$
$$= \{GStoPV \mapsto \langle\, MemOk, ?SVto(pid), !(pid)toSV\,\rangle, GStoHS_2 \mapsto \langle \texttt{HSAlloc}\rangle\}$$
$$\circ\,(h(rt^k.(q \circ freesr), rt^k.(q \circ freeb)) \otimes (f_{A_{pid}}(\texttt{chA}) \otimes f_{LRU_{pid}}(\texttt{chLRU})))$$
$$(\{GSto(pid) \mapsto \langle \texttt{SVAlloc}\rangle, (pid)toL \mapsto \langle \texttt{LAlloc}\} \circ s)$$

wobei $\texttt{HSAlloc} = Alloc[\ ((q \circ freeb)[1], slist[1]),\ \ \ldots, ((q \circ freeb)[k], slist[k])\ ]$
$\texttt{LAlloc} = (p \circ freesr)[1] \circ \ldots \circ (p \circ freesr)[k]$
$\texttt{SVAlloc} \triangleq (k, (q \circ freeb)[1] \circ \ldots \circ (q \circ freeb)[k])$

wobei : $k = \#slist,\ \texttt{chLRU} = ?(pid)toL, !Lto(pid),$
$\texttt{chA} = ?GSto(pid), !(pid)toGS, ?(pid)toSV, !SVto(pid), ?Lto(pid), !(pid)toL$

7.8 Terminierende Komponenten

Der letzte Schritt unserer Modellierung rundet die Durchführung von Berechnungen durch Prozesse ab. Die Komponenten, die im System aufgrund eines Benutzerauftrags erzeugt wurden, müssen wieder aus dem System entfernt werden, sobald die Auftragsausführung beendet ist. Die Komponenten, die erzeugt wurden, sind die lokalen Speicherverwalter und die Prozesse. Aus diesem Grund werden wir die Spezifikationen der Komponenten Pi und SV_{Pi} in den folgenden beiden Abschnitten abschließend um die entsprechende Formalisierung erweitern.

7.8.1 Prozesse und Prozessor

In Abschnitt 4.3.3 wurde mit den Gleichungen (5a) und (5b) bereits der Fall behandelt, in dem das Programm vollständig abgearbeitet wurde. In Abschnitt 2.6 von Kapitel 2 wurde beschrieben, daß in FOCUS eine Komponente durch das Löschen ihrer gesamten Schnittstelle aus dem System entfernt wird.

Entsprechend zur impliziten Modellierung des Zustandes *started* werden wir im folgenden auch die Terminierung eines Prozesses modellieren. Der in Abschnitt 4.3.3 auf Seite 57 textuell erklärte und auf Seite 4.3.3 formalisierte Punkt (5) wird ersetzt. Zunächst geben wir die textuelle Beschreibung:

(5) Erhält Pi das „Ok“ zum letzten Berechnungsschritt, sendet er die Nachricht *Term* über Kanal $PtoPZ$ und löscht die Verbindungen zu PZ. Zusätzlich sendet $P1$ ein $\langle OutputPi \rangle$ über Kanal Out_i und löscht seine gesamte Schnittstelle zur Umgebung und zur Prozessorverwaltung.

(11) *Sendeoperation bei Rendezvous*
Pi erhält im Zustand *waitKoop* über Kanal $ConftoPi$ die Nachricht $Conf$ und den Port $!ConftoPi$. Es gilt $\#p = 1$. Pi sendet die Nachricht $OutputPi$ über Kanal Out_i und terminiert.

Diese Beschreibung wird in folgende Spezifikation umgesetzt, wobei wir beispielhaft davon ausgehen, daß Pi der Familie F_k zugeordnet ist:

Anpassung der Spezifikation für f_{Pi} von Seite 57

Für $\#p = 1$:

$$(5a)\quad h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p)\rangle\} \circ s) = \{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ\rangle, Out_i \mapsto \langle OutputPi, \boldsymbol{?In_i, !Out_i}\rangle, \boldsymbol{PitoQ} \mapsto \langle \boldsymbol{!PitoQ}\rangle\} \circ \mathtt{null}$$

Für Kooperation mit Rendezvous:

$$(5b)\quad h(busy, p)(\{PZtoP \mapsto \langle Ok(ft.p), Suspend, \mathtt{InChanList}\rangle\} \circ s) = \{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ\rangle, \mathtt{Term}\} \circ \mathtt{null}$$

$$(11)\quad h(waitKoop, Send(Val, Pj))(\{ConftoPi \mapsto \langle Conf, !ConftoPi\rangle\} \circ s) = \{\mathtt{Term}\} \circ \mathtt{null}$$

wobei: $\mathtt{InChanList} = ?SVtoPi, !PitoSV, ?KV_{Fk}toPi, !PitoKV_{Fk}$

$$\mathtt{Term} = Out_1 \mapsto \langle OutputP1, ?In_i, !Out_i\rangle, PitoQ \mapsto \langle !PitoQ\rangle, PitoSV \mapsto \langle ?SVtoPi, !PitoSV\rangle, PitoKV_{Fk} \mapsto \langle ?KV_{Fk}toPi, !PitoKV_{Fk}\rangle$$

Diese Gleichungen ergeben sich direkt mit den Erläuterungen aus Abschnitt 2.6 zum Löschen von Komponenten. Damit in Fall (5a) die Verbindungen zum lokalen Speicherverwalter und dem Kooperationsmanager gelöscht werden, ersetzen wir die Gleichungen (5) und (6) der Modellierung des Prozessors von Seite 60 durch folgende Spezifikation:

Anpassung der Spezifikation für f_{PZ} von Seite 60

(5) $f_{PZ}(\{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ \rangle, DtoPZ \mapsto t\} \circ s)$
$= \{PZtoD \mapsto \langle Term, ?!PtoPZ, ?!PZtoP \rangle, \boldsymbol{PitoSV} \mapsto \langle \boldsymbol{?SVtoPi, !PitoSV} \rangle,$
$\boldsymbol{PitoKV_{Fk}} \mapsto \langle \boldsymbol{?PitoKV_{Fk}, !KV_{Fk}toPi} \rangle\} \circ f_{PZ}(s)$

(6) $f_{PZ}(\{PtoPZ \mapsto \langle Term, ?PZtoP, !PtoPZ \rangle, DtoPZ \mapsto \langle Suspend \rangle\} \circ s)$
$= \{PZtoD \mapsto \langle ?!PtoPZ, ?!PZtoP \rangle, \boldsymbol{PitoSV} \mapsto \langle \boldsymbol{?SVtoPi, !PitoSV} \rangle,$
$\boldsymbol{PitoKV_{Fk}} \mapsto \langle \boldsymbol{?PitoKV_{Fk}, !KV_{Fk}toPi} \rangle\} \circ f_{PZ}(s)$

7.8.2 Lokale Speicherverwalter

Abschließend beschreiben wir die Terminierung eines lokalen Speicherverwalters beispielhaft für einen Prozeß Pi. Auch eine Komponente SV_{Pi} wird aus dem System entfernt, indem sie ihre gesamte Schnittstelle löscht und somit nicht mehr in das System eingebunden ist. Beim Löschen eines lokalen Speicherverwalters muß zusätzlich berücksichtigt werden, daß er die Seitenrahmen und Blöcke, die der Prozeß mit seiner Terminierung frei gegeben hat, wieder an den globalen Speicherverwalter zurückgibt. Zu der Modellierung eines lokalen Speicherverwalters aus Abschnitt 5.5.1 nehmen wir die folgende Gleichung hinzu:

(6) Erhält A_{Pi} über Kanal $PitoSV$ die Ports $?SVtoPi$ und $!PitoSV$, so registriert A_{Pi}, daß Pi terminiert ist. A_{Pi} sendet über Kanal $PitoGS$ jeweils eine Liste von Identifikatoren der Blöcke und Seitenrahmen, die Pi belegt hat und löscht die Schnittstelle zu GS und zu LRU_{Pi}.

Wir erhalten folgende Formalisierung:

Erweiterung der Spezifikation für $f_{A_{Pi}}$ von Seite 121

(6) $g(z)(\{PitoSV \mapsto \langle ?SVtoPi, !PitoSV \rangle\} \circ s)$
$= \{PitoGS \mapsto \langle sr, b, ?SVtoPi, !PitoSV \rangle, PitoL \mapsto \langle ?LtoPi, !PitoL \rangle\} \circ \texttt{null}$

wobei: $sr = SRid©(z[0] \circ \ldots \circ z[n_{Pi}])$ und $b = Bid©(z[0] \circ \ldots \circ z[n_{Pi}])$

Kapitel 8

Zusammenfassung und Ausblick

Mit der formalen Modellierung von Betriebssystemkonzepten wurde ein Brückenschlag zwischen Focus und einem komplexen Anwendungsgebiet aus der praktischen Informatik vorgenommen. Die Ergebnisse der Arbeit sind in erster Linie aus der Sicht von Focus

- als Anwendung zur formalen Spezifikationsentwicklung,
- als Demonstration zur Praxistauglichkeit und
- als Entwicklung methodischer Anleitungen

zu sehen. Im vorliegenden Kapitel werden die Ergebnisse der Arbeit zusammengefaßt und bewertet. Abschließend geben wir erste verallgemeinernde Hinweise zur methodischen Vorgehensweise bei der Erstellung von Spezifikationen in dem in der vorliegenden Arbeit gewählten Stil und Anregungen für weiterführende Arbeiten.

8.1 Modellierung eines Betriebssystems

Zur Weiterentwicklung ausgereifter formaler Methoden ist es wesentlich, deren Praxistauglichkeit anhand großer Anwendungen zu überprüfen und nachzuweisen. Mit dem Vorsatz, eine Spezifikationsentwicklung für Focus systematisch und unter methodischen Gesichtspunkten durchzuführen, ergab sich die in Abschnitt 1.1 beschriebene Aufgabenstellung für die vorliegende Arbeit. Das erzielte Ergebnis läßt sich wie folgt zusammenfassen:

> Unter Verwendung von Focus wurde die formale Modellierung eines Systems entwickelt, dessen Verhalten der Funktionsweise wesentlicher Teile eines Betriebssystems auf hohem Abstraktionsniveau entspricht. Mit dem Einsatz speziell entwickelter Spezifikationsschemata konnten die Formalisierungen systematisch erstellt werden. Durch die gewählte methodische Vorgehensweise ergibt sich die vollständige Systemspezifikation in kleinen nachvollziehbaren Schritten, die an Teilfunktionalitäten eines Betriebssystems orientiert sind.

Vor dem Hintergrund, daß ein Betriebssystem das *Management* eines Rechensystems darstellt, ergeben sich die verschiedenen Teilaufgaben der Ressourcenverwaltung, durch die der Aufbau der vorliegenden Arbeit motiviert und die grundlegende methodische Vorgehensweise geprägt wurden. Ein Betriebssystem ist dafür zuständig, daß Aufträge, die Benutzer an ein Rechensystem erteilen, vollständig durchgeführt und mit dem gewünschten Ergebnis abgeschlossen werden. Hierfür wird die Hardware eines Rechensystems durch das Betriebssystem entsprechend nutzbar gemacht. Die vier zentralen Bestandteile der Betriebsmittelverwaltung umfassen die Prozessor- und Speicherverwaltung sowie die Prozeßkooperation und das Erzeugen bzw. Auflösen von Prozessen. Der strukturelle Aufbau des Systems, dessen Spezifikation in der vorliegenden Arbeit entwickelt wurde, ist in Abbildung 8.1.1 dargestellt. Hierbei zeigen wir den Aufbau des Systems mit einem beispielhaft dargestellten Prozeß Pi und verzichten auf die Darstellung weiterer Prozesse sowie spezieller Teilkomponenten der Subsysteme, die die oben genannten Verwaltungsaufgaben gewährleisten. Die graphische Darstellung dieser Subsysteme ist jeweils mit einem Verweis auf das Kapitel versehen, in dem die formale Modellierung entwickelt wurde.

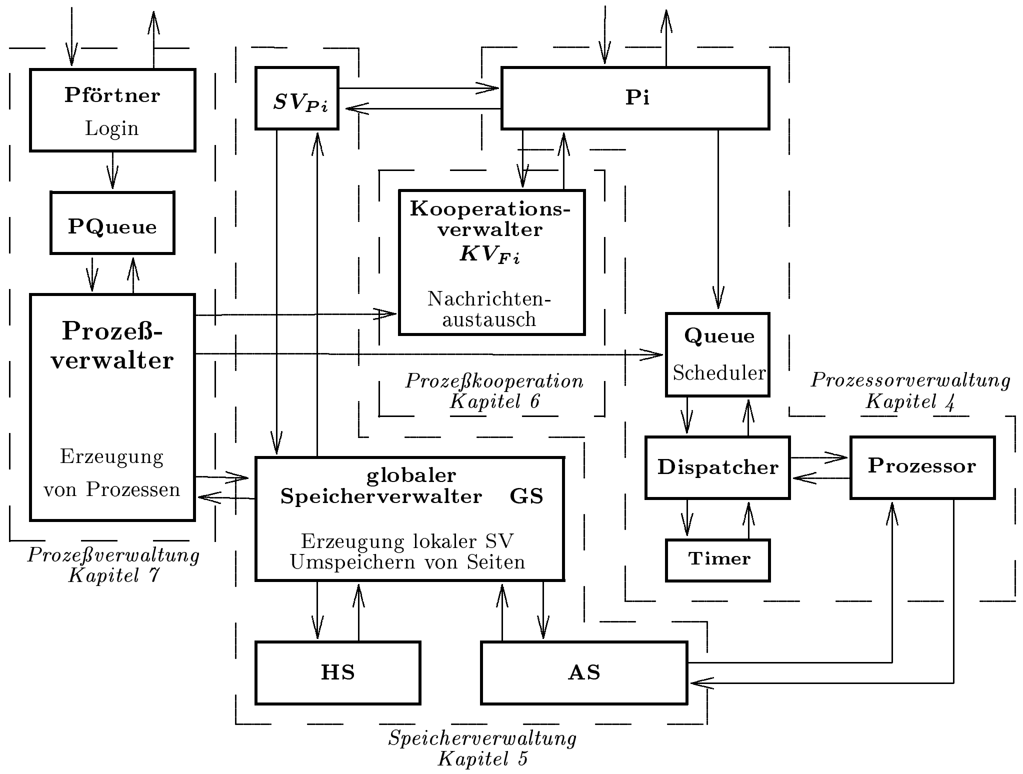


Abbildung 8.1.1: Ein mit FOCUS modelliertes Betriebssystem

Die Entwicklung einer Spezifikation eines Betriebssystems erfolgt durch Erweiterungen, Anpassungen und Ergänzungen der Spezifikation des *Kernsystems* zur Prozessorverwaltung. Die methodische Vorgehensweise orientiert sich an den vier genannten Bereichen der Ressourcenverwaltung. Durch die Modellierung auf hohem Abstraktionsniveau und Erarbeitung wesentlicher Charakteristika einzelner Algorithmen und Verfahren werden Betriebssystemkonzepte in kleinen Schritten und anhand der entwickelten FOCUS-Spezifikation erklärt. Insbesondere wird die Analogie zwischen Betriebssystemprozessen und Prozessoren mit jedem Entwicklungsschritt deutlicher, da die Hinzunahme von Verwaltungsaufgaben die entsprechende Erweiterung des Verhaltens beider Komponenten zur Folge hat.

- Das Verhalten eines *Prozesses* wird durch ein Zustandsdiagramm veranschaulicht, siehe Abbildung 7.2.1 auf Seite 159. Die Zustände und Zustandsübergänge entsprechen den Schritten, die allgemein für alle Prozesse aus Betriebssystemsicht und abstrahiert von ihrem jeweiligen Programm durchgeführt werden.

- Das Verhalten eines *Prozessors* ist dadurch charakterisiert, daß er die Instruktionen von Berechnungen ausführt. Die Ausführung *einer* Instruktion besteht letztlich aus einer Folge von Schritten, die zur Nutzung der Ressourcen erforderlich sind.

Die Spezifikation eines Prozesses wird schrittweise um weitere Zustände ergänzt, und analog dazu wird die Ausführung einer Instruktion durch den Prozessor in neue Teilschritte aufgebrochen. Diese Modellierung liefert eine Veranschaulichung, die in der Betriebssystemliteratur in dieser Form nicht auftritt, aber charakteristisch für Betriebssysteme ist. Insgesamt werden wesentliche Charakteristika, Aufgaben und Bestandteile eines Betriebssystems auf hohem Abstraktionsniveau erläutert. Diese abstrakte Beschreibung geht mit der Spezifikationserstellung Hand in Hand und liefert eine einfache und gut verständliche Erklärung der in Betriebssystemen auftretenden Managementaufgaben.

8.2 Ergebnisse zur methodischen Vorgehensweise

Viele der bisher mit FOCUS behandelten Beispiele sind zwar aus theoretischer Sicht interessant, haben jedoch aus Anwendersicht oftmals zu wenig Bezug zu *realen* Problemen. Mit der vorliegenden Arbeit wurde die Spezifikation eines Systems entwickelt, dessen Verhalten an einer Standardanwendung der praktischen Informatik, den Betriebssystemen, orientiert ist. Es wurde gezeigt, daß FOCUS zur formalen Modellierung einer Anwendung dieser Komplexität gut geeignet ist. Vor allem die erweiterte semantische Basis von FOCUS zur Spezifikation von Systemen, deren Struktur sich während eines Systemablaufs verändern kann, hat sich als wesentlich erwiesen. Das Erzeugen und Auflösen von Prozessen, und vor allem die explizite Modellierung der Zuteilung und des Entzugs von Betriebsmitteln wäre mit einer statischen Systemstruktur nur mit großem Aufwand möglich gewesen.

Neben der Anwendung von FOCUS auf eine komplexe, reale Aufgabenstellung sollte ein möglicher Weg aufgezeigt werden, *wie* Formalisierungen in FOCUS erstellt und Spezifikationen systematisch und schrittweise entwickelt werden können. Hierfür haben wir zunächst einen Spezifikationsstil festgelegt, der in der Arbeit zur Verhaltensbeschreibung eingesetzt wird: Funktionsgleichungen und den mathematisch-logischen Stil. Dieser operationale Stil entspricht folgender, für Verhaltensbeschreibungen in FOCUS, zentralen Vorstellung: Eine Komponente liest auf ihren Eingabekanälen die gemäß der festgelegten Nachrichtenmengen gültigen Eingabemuster. Die Spezifikation legt fest, welche Ausgabemuster die Komponente als Reaktion über ihre Ausgabekanäle sendet. Für die Kanäle werden mögliche Eingabemuster bestimmt und Spezifikationen als Mengen von Funktionsgleichungen erstellt. Diese Vorgehensweise entspricht in gewisser Weise der Erstellung eines abstrakten Programms. Die Spezifikationen werden in einer einheitlichen Form erstellt, wobei einige wenige festgelegte Notationen eingehalten werden müssen.

Zur Umsetzung dieses operationalen Verständnisses einer Verhaltensbeschreibung wurden *Spezifikationsschemata* festgelegt, die es ermöglichen, das geforderte Verhalten zunächst in textueller Form zu entwickeln und die textuelle Beschreibung dann in eine Formalisierung umzusetzen. Der angegebene Katalog der Schemata, vor allem für mobile, dynamische Systeme, entstand Hand in Hand mit der Entwicklung der Spezifikationen und hat sich als ausreichend erwiesen, um alle hier gezeigten Formalisierungen zu erstellen. Eine derartig aufbereitete Anleitung zur Spezifikationsentwicklung lag bisher für FOCUS nicht vor. Die Anleitung für den Einsatz der Funktionsgleichungen kann für FOCUS allgemein und unabhängig von der Modellierung von Betriebssystemkonzepten eingesetzt werden.

Die insgesamt beschrittene methodische Vorgehensweise wurde ausschließlich für die Modellierung der Betriebssystemkonzepte festgelegt und hier angewendet. Sie läßt sich wie folgt allgemein charakterisieren:

> Die Weiterentwicklung einer Formalisierung orientiert sich an der wachsenden Komplexität der Aufgabenstellung. Aufgrund vorbereitender Maßnahmen in *einfachen* Spezifikationen entstehen deren Erweiterungen durch Anpassungen und Vervollständigungen. Die formalen Modellierungen der Systeme mit wachsender Komplexität sind schrittweise nachvollziehbar und erscheinen dadurch leichter verständlich, als die direkte Modellierung des Systems in seinem vollen Umfang.

Die Erstellung von Spezifikationen für Systeme mit komplexem Verhalten bleibt prinzipiell schwierig, und der Umfang einer Spezifikation kann trotz starker Abstraktion nicht beliebig reduziert werden. Dennoch können wir folgende allgemeine Hinweise geben:

- Eine komplexe Aufgabenstellung sollte in kleine handhabbare Teile mit reduzierter Komplexität aufgeteilt werden.
- Das zu spezifizierende Verhalten sollte zunächst auf die wesentliche Kernfunktionalität des Systems reduziert werden. Ausgehend von diesem *Kernsystem* wird die vollständige Spezifikation mit der Hinzunahme weiterer Funktionalitäten, die auf die formalen Spezifikationen übertragen werden, schrittweise entwickelt. Ob die Reduktion auf ein derartiges Kernsystem immer so erfolgen kann, daß diese Vorgehensweise von Vorteil ist, muß anhand weiterer Anwendungen überprüft werden.
- Komponenten mit einer großen Zahl von Ein- und Ausgabekanälen, die viele Informationen erhalten, verarbeiten oder möglicherweise speichern müssen, sollten so weit

wie möglich in kooperierende Subkomponenten aufgebrochen werden, die für einzelne Teilaufgaben zuständig sind und deren Spezifikation handhabbar ist.

Ausgehend von unseren Erfahrungen mit der speziellen Anwendung schlagen wir folgende Vorgehensweise zur Erstellung von Spezifikationen im hier gewählten Spezifikationsstil vor:

1. Ausgehend von dem grob festgelegten Verhalten müssen der strukturelle Aufbau des verteilten Systems konzipiert sowie die Komponenten und deren Abhängigkeiten festgelegt werden. Die Erarbeitung dieser Konzeption anhand einer graphischen Darstellung liefert ein erstes Design für den Systemaufbau mit klaren Beschreibungsmitteln, die wie in unserer Anwendung die SSDs sogar die Definition der Schnittstellen und die für eine ANDL-Spezifikation benötigten Festlegungen liefern.

2. Das im Groben festgelegte Verhalten wird auf Ein- und Ausgabemuster sowie deren Beziehung zueinander übertragen. Diese Zuordnungen werden in textueller und strukturierter Form beschrieben. Da hierbei keine Formalisierung erstellt wird, ist es möglich, das geforderte Verhalten zu erarbeiten, ohne spezielle *formale* Notationen verwenden zu müssen.

3. Wurde die textuelle Beschreibung gemäß der Schemata erstellt, so ist die Formalisierung bei der Entwicklung der Verhaltensspezifikation ein nachgeordneter Schritt.

4. Die Erstellung einer *mathematisch korrekten* Spezifikation, in der alle Variablen definiert und quantifiziert sind, erfolgt im letzten und abschließenden Schritt: Die Typen aller verwendeten Variablen werden festgelegt. Um eine Spezifikationen geeignet zu strukturieren, können Nachrichtenmuster, die in mehreren Gleichungen auftreten, durch abkürzende Bezeichner ersetzt werden, die formal zu definieren sind.

5. ANDL-Spezifikationen können sowohl für verteilte Systeme als auch für Komponenten direkt aus SSDs hergeleitet werden. In unserer Modellierung liefert eine ANDL-Spezifikation die semantische Basis. Aufgrund einer einmal geeignet getroffenen Festlegung werden semantische Details verborgen. Spezifikationen, die in einem so festgelegten Rahmen erstellt werden, sind semantisch fundiert.

8.3 Weiterführende Arbeiten

Im folgenden gehen wir auf weiterführende Arbeiten ein, die sich als Weiterführung der in der Arbeit erzielten Ergebnisse anbieten.

In der vorliegenden Arbeit wurde zur Spezifikation i.w. *eine* spezielle Beschreibungstechnik eingesetzt und so aufbereitet, daß Formalisierungen schematisch erstellt werden können. Insgesamt sollten für *alle* Beschreibungstechniken von FOCUS Anleitungen für deren Einsatz und Entscheidungskriterien zur Auswahl erarbeitet werden. Die mit FOCUS vorgegebene *Methodik im Großen* sollte so aufbereitet werden, daß festgelegte Richtlinien und Vorgaben zur Durchführung einer Systementwicklung zur Verfügung stehen. Ein derartiges

Handbuch bietet einem Anwender die Möglichkeit, die für seinen Anwendungsfall passende Beschreibungstechnik und Vorgehensweise zu wählen.

Bei der Entwicklung von Systemen mit einer großen Anzahl von Komponenten tritt oftmals der Fall ein, daß Komponenten mit gleicher oder zumindest sehr ähnlicher Funktionalität mehrfach auftreten. In unserer Modellierung ist die an vielen Stellen eingesetzte *Queue* hierfür ein Beispiel. Für einen komfortablen Einsatz von Focus würde sich in diesem Fall das Konzept der Wiederverwendung anbieten. In einer Bibliothek stehen vorbereitete Spezifikationen von Komponenten und Netzen mit spezifischer Funktionalität zur Verfügung. Auf diese kann beispielsweise durch eine systematische Parametrisierung gezielt zugegriffen werden.

Es zeigt sich, daß graphische Beschreibungstechniken zur Konzeption und Veranschaulichung von Modellierungen für große Anwendungen gut geeignet sind. SSDs wurden in der gesamten Arbeit zur Konzeption des strukturellen Aufbaus der Teilssysteme und damit der Abhängigkeiten zwischen Komponenten sowie zur Veranschaulichung des schrittweisen Aufbaus der Modellierung eingesetzt. In den SSDs konnten wir zwar mit speziellen Notationen arbeiten und so auf die Darstellung von Komponenten mit *gleicher* Funktionalität verzichten, dennoch wurde die Darstellung teilweise komplex. Hier sollten zukünftige Arbeiten darauf abzielen, vor allem für variable Systemstrukturen, Erweiterungen der SSDs und entsprechender Beschreibungstechniken bereitzustellen.

Zur Spezifikation wurden SSDs *und* Andl verwendet. Andl wurde so konzipiert, daß eine Spezifikation aus einem SSD hergeleitet werden kann und die Fundierung in Holcf liefert. Entsprechend dazu wurde Andl in unserer Arbeit verwendet: Ein SSD liefert die Veranschaulichung der Systemstruktur und Andl, in Verbindung mit den Funktionsgleichungen, die Semantik der Spezifikation. Der konsequente Einsatz beider Beschreibungstechniken führt jedoch teilweise dazu, daß gleiche Information (Kanal- und Komponentenbezeichner) mehrfach und redundant auftritt. Dies kann behoben werden, indem eine Darstellung als SSD sofort auch die semantische Fundierung der Spezifikation liefert. Dieser Schritt ist in Kombination mit den bereits genannten Arbeiten zur Entwicklung der Beschreibungstechniken für große Systeme mit variabler Systemstruktur zu sehen.

Eine weitere Möglichkeit, Funktionsgleichungen zu notieren, ist in Focus durch den tabellarischen Stil gegeben. Hierbei werden die Ein- und Ausgabemuster in Tabellen und dort in Spalten, die den Ein- bzw. Ausgabekanälen zugeordnet sind, festgehalten. Zur Spezifikation eines Systems mit statischer Struktur, also Systeme deren Anzahl der Ein- und Ausgabekanäle für alle Systemabläufe fest vorgegeben ist, ergibt sich hier eine einheitliche Darstellung. Für Systeme mit variabler Schnittstelle kann keine entsprechende feste Darstellung zur Dokumentation der Kanäle angegeben werden, die zur aktuellen Schnittstelle gehören. Eine Spezifikation kann beispielsweise durch mehrere Tabellen erstellt werden, die jeweils für eine Systemphase mit fester Schnittstelle gültig sind, siehe ein erster Ansatz in [HS96]. Im von uns gewählten mathematischen Stil ist diese Dokumentation in der Semantik verborgen und muß nicht explizit aufgeführt werden, so daß sich wohlstrukturierte Spezifikationen ergeben. Eine Verallgemeinerung des tabellarischen Spezifikationsstils auch für variable Systemstrukturen sollte jedoch für Focus erarbeitet werden, wobei eine Kombination mit dem mathematisch-logischen Stil durchaus von Vorteil wäre.

In der vorliegenden Arbeit stand die *Modellierung* eines verteilten Systems im Vordergrund, mit dem wesentliche Teile eines Betriebssystems beschrieben werden. Die *Entwicklung* eines Betriebssystems als eine aus der Spezifikation hergeleitete Implementierung war *nicht* vorgesehen. Viele der hierfür wichtigen realisierungstechnischen Eigenschaften wurden nicht berücksichtigt. Vor diesem Hintergrund ist für die Entwicklung einer Implementierung nur ein allererster Schritt getan. Die Entwicklung hoch qualitativer Betriebssysteme sollte jedoch unter Verwendung formaler Methoden und in Top-Down-Vorgehensweise erfolgen, wofür mit der erstellten formalen Spezifikation ein Grundstein gelegt wurde. In bezug auf die Entwicklung eines Betriebssystems sind somit viele weiterführende Arbeiten denkbar.

Der zunehmende Einsatz von Computern in verschiedensten Bereichen, wie beispielsweise in Kraftfahrzeugen, Haushaltsgeräten oder mobilen Telefonen, hat dazu geführt, daß Software benötigt wird, die in speziellen Geräten eingesetzt werden und deren Funktionalität der eines Betriebssystems mit eingeschränkter Funktionalität entspricht. Vor allem hier wird es unerläßlich sein, Standards zu entwickeln, an die hohe Sicherheits- und Qualitätsanforderungen gestellt werden. Der Einsatz formaler Methoden zur vollständigen Beschreibung und Entwicklung derartiger *kleiner* Betriebssysteme mit spezifischer Funktionalität erscheint hier besonders reizvoll.

Im Hinblick auf die Steigerung der Qualität von Betriebssystemen ist es von besonderem Interesse, spezifische und von einer Implementierung unabhängige Eigenschaften zu gewährleisten. Beispiele hierfür sind:

- Die Garantie dafür, daß jeder Prozeß mit der Ausgabe des gemäß der Berechnungsvorschrift korrekten Ergebnisses terminiert und somit Benutzeraufträge (Berechnungen) korrekt und zuverlässig ausgeführt werden.
- Die Sicherstellung der Deadlock-Freiheit bei kooperierenden Prozessen.
- Die Gewährleistung dafür, daß Abhängigkeiten von Prozessen zu deren ordnungsgemäßer Terminierung führen: ein Prozeß darf erst dann terminieren, wenn alle Prozesse, von denen er abhängig ist, terminiert sind.

Derartige Eigenschaften sind für ein System nur dann von Nutzen, wenn ihre Gültigkeit durch formale Beweise sichergestellt werden kann. Beweise können jedoch erst auf der Basis einer fundierten formalen Spezifikation geführt werden. Diese steht mit der in der vorliegenden Arbeit für wesentliche Teile eines Betriebssystems auf hohem Abstraktionsniveau und unabhängig von technischen Details erstellten Modellierung für zukünftige und weiterführende Arbeiten zur Verfügung.

Literaturverzeichnis

[AMST92] G. Agha, I. A. Mason, S. F. Smith und C. L. Talcott. *Towards a Theory of Actor Computation.* in *The Third International Conference on Concurrency Theory (CONCUR '92)*, Lecture Notes in Computer Science Bd. 630. Springer 1992, S. 565–579.

[BA81] J.D. Brock und W.B. Ackermann. *Scenarios: A Model of Nondeterministic Computation.* Foundations of Programming Concepts (1981), 252–259.

[BD91] M. Broy und C. Dendorfer. *Functional Modelling of Operating System Structures by Timed Higher Order Stream Processing Functions (revised version).* SFB-Bericht 342/22/90 A, Technische Universität München, 1991.

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner und R. Weber. *The Design of Distributed Systems — An Introduction to* FOCUS. SFB-Bericht 342/2/92 A, Technische Universität München, 1992.

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner und R. Weber. *The Design of Distributed Systems – An Introduction to* FOCUS. SFB-Bericht Nr. 342/2-2/92 A, Technische Universität München, 1993.

[BHL+96] J. P. Bowen, C. A. R. Hoare, H. Langmaack, E.-R. Olderog und A. P. Ravn. *A ProCoS II Project Final Report: ESPRIT Basic Research Project 7071.* Bulletin of the European Association for Theoretical Computer Science (EATCS) **59** (Juni 1996).

[BHS96] M. Broy, H. Hußmann und B. Schätz. *Formal Development of Consistent System Specifications.* in Marie-Claude Gaudel und James Woodcock (Hrsg.), *FME'96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science Bd. 1051. Springer 1996, S. 248–267.

[BMS96a] M. Broy, S. Merz und K. Spies (Hrsg.). *Formal Systems Specification – The RPC-Memory Specification Case Study*, Lecture Notes in Computer Science Bd. 1169. Springer 1996.

[BMS96b] M. Broy, S. Merz und K. Spies. *The RPC-Memory Specification Problem: A Synopsis.* in *[BMS96a]*, Lecture Notes in Computer Science 1169. Springer 1996, S. 5–20.

[Bre97] M. Breitling. *Formalizing an Verifying* TIMEWARP *with* FOCUS. SFB-Bericht 342/27/97 A, Technische Universität München, 1997.

[Bro87] M. Broy. *Semantics of Finite and Infinite Networks of Concurrent Communicating Agents.* Distributed Computating **2 (1)** (1987), 13–31.

[Bro92a] M. Broy. *Compositional Refinement of Interactive Systems.* Technical Report 89, Digital Systems Research Center, Pallo Alto, California 94301 1992.

[Bro92b] M. Broy. *(Inter)-Action Refinement: The Easy Way - Compositional refinement of interactive systems.* Technical Report 10, International Summer School, Marktoberdorf 1992. Also appears in the corresponding proceedings.

[Bro94] M. Broy. *A Functional Rephrasing of the Assumption/Commitment Specification Style.* SFB-Bericht 342/10/94 A, Technische Universität München, 1994.

[Bro95a] M. Broy. *Equations for Describing Dynamic Nets of Communicating Systems* . in A. Tarlecki E. Astesiano, G. Reggio (Hrsg.), *Recent Trends in Data Types Specification, 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop*, Lecture Notes of Computer Science Bd. 906. Springer 1995, S. 170–187.

[Bro95b] M. Broy. *A Functional Specification of the Alpha AXPTM Shared Memory Model.* SRC Research Report 136, DIGITAL Systems Research Center, 1995.

[Bro96] M. Broy. *A Functional Solution to the RPC-Memory Specification Problem.* in *[BMS96a]*, Lecture Notes in Computer Science Bd. 1169. Springer 1996, S. 183–212.

[Bro97] M. Broy. *Refinement of Time.* in *Proceedings of Fourth AMAST Workshop on Real-Time Systems, Concurrent, and (ARTS'97)*, Lecture Notes in Computer Science, Springer, Palma de Mallorca 1997. to appear.

[BS98] M. Broy und K. Stølen. *FOCUS on System Development – A Method for the Development of Interactive Systems*, 1998. Manuskript.

[CD73] E.G. Coffman und P.J. Denning. *Operating Systems Theorie.* Prentice-Hall, 1973.

[CM88] K.M. Chandy und J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[Den95] C. Dendorfer. *Methodik funktionaler Systementwicklung.* Dissertation, Technische Universität München, 1995.

[Fin88] R.A. Finkel. *An Operating Systems Vade Mecum.* Prentice Hall, 1988.

[FS93] M. Fuchs und K. Stølen. *Development of a Distributed Min/Max Component.* SFB-Bericht 342/18/93 A, Technische Universität München, 1993.

[Fuc94] M. Fuchs. *Technologieabhängigkeit von Spezifikationen digitaler Hardware.* SFB-Bericht 342/14/94 A, Technische Universität München, 1994. Dissertation.

[GKRB96] R. Grosu, C. Klein, B. Rumpe und M. Broy. *State Transition Diagrams.* TUM-I 9630, Technische Universität München, 1996.

[GO95] J. Gulbins und K. Obermayer. *UNIX System V.4.* Springer, 1995.

[Gro94] R. Grosu. *Concurrent Object Oriented Programming.* Dissertation, Technische Universität München, 1994.

[Gro96a] R. Grosu. *Banken und Konten: Ein Beispiel zur Spezifikation nebenläufiger Informationssysteme*, 1996. Interne Ausarbeitung.

[Gro96b] R. Grosu. *Recursive Networks and Time Abstraction*, 1996. Interne Ausarbeitung.

[GS94] Peter B. Galvin und A. Silberschatz. *Operating System Concepts, 4th ed.* Addison-Wesley, 1994.

[GS96a] R. Grosu und K. Stølen. *A Model for Mobile Point-to-Point Data-flow Networks without Channel Sharing.* in M. Nivat M. Wirsing (Hrsg.), *AMAST'96*, Lecture Notes in Computer Science Bd. 1101. Springer 1996.

[GS96b] R. Grosu und K. Stølen. *A Denotational Model for Mobile Many-to-Many Dataflow Networks.* Technischer Bericht TUM-I9622, Technische Universität München, 1996.

[GS96c] R. Grosu und K. Stølen. *A Denotational Model for Mobile Point-to-Point Dataflow Networks with Channel Sharing.* Technischer Bericht, Technische Universität München, 1996.

[GS97] R. Grosu und K. Stølen. *Compositional Specification of Mobile Systems.* Formal Aspects of Computing (1997). to appear.

[HBS73] C. Hewitt, P. Bishop und R. Steiger. *A Universal Modular Actor Formalism for Artificial Intelligence.* in *Proc. IJCAI'73*, Morgan Kaufmann Publishers, Standford, California 1973, S. 235–245.

[Hin96] U. Hinkel. *Einbettung von SDL in Logik (Teil 2)*, 1996. Interner Bericht.

[Hin97] U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL*, 1997. Dissertation, *vorläufige Version.*

[HS96] U. Hinkel und K. Spies. *Anleitung zur Spezifikation von mobilen, dynamischen* FOCUS-*Netzen.* SFB-Bericht 342/16/96 A, Technische Universität München, 1996.

[HS97] U. Hinkel und K. Spies. *Spezifikationsmethodik für mobile, dynamische FOCUS-Netze.* in Adam Wolisz, Axel Rennoch und Ina Schieferdecker (Hrsg.), *Formale Beschreibungstechniken für verteilte Systeme*, GMD - Studien (ISSN 0170-8120). GMD - Forschungszentrum Informationstechnik, Sankt Augustin 1997, S. 251–261.

[HSE97] F. Huber, B. Schätz und G. Einert. *Consistent Graphical Specification of Distributed Systems.* in Cliff B. Jones Peter Lucas, John Fitzgerald (Hrsg.), *FME'97: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science Bd. 1313. Springer 1997, S. 122–141.

[HSS96] F. Huber, B. Schätz und K. Spies. *AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme.* in Ulrich Herzog Holger Hermanns (Hrsg.), *Formale Beschreibungstechniken für verteilte Systeme.* Universität Erlangen-Nürnberg 1996, S. 165–174. Erschienen in: Arbeitsbereichte des Insituts für mathematische Maschinen und Datenverarbeitung, Bd.29, Nr.9.

[Int96] International Telecommunication Union, Geneva. *Message Sequence Charts*, 1996. ITU-T Recommendation Z.120.

[IT93] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL).* ITU, 1993.

[JS89] S.B. Jones und A.F. Sinclair. *Functional Programming and Operating Systems.* The Computer Journal **32, No.2** (1989), 162–174.

[Kah74] G. Kahn. *The Semantics of a Simple Language for Parallel Programming.* Information Processing (1974), 471 – 475.

[Lam94] L. Lamport. *The Temporal Logic of Actions.* ACM TRansactions on Programming Languages and Systems **16** (1994) 3, 872–923.

[MOO87] M. Maekawa, A.E. Oldehoeft und R.R. Oldehoeft. *Operating Systems – Advanced Concepts.* The Benjamin/Cummings Publishing Company, 1987.

[MPW92a] R. Milner, J. Parrow und D. Walker. *A calculus of mobile processes, part I.* Information and Computation **100** (1992), 1–40.

[MPW92b] R. Milner, J. Parrow und D. Walker. *A calculus of mobile processes, part II.* Information and Computation **100** (1992), 41–77.

[Ohe95] D. von Oheimb. *Datentypspezifikationen in Higher-Order LCF.* Diplomarbeit, Technische Universität München, 1995.

[PS85] J.L. Peterson und A. Silberschatz. *Operating System Concepts.* Addison-Wesley, 1985.

[San96] R. Sandner. *Unterstützung von Strukturverfeinerungen in FOCUS durch Isabelle - Verifikation einer Fertigungszelle.* Diplomarbeit, Technische Universität München, 1996.

[Slo97] O. Slotosch. *Refinements in* HOLCF*: Implementation of Interactive Systems.* Dissertation, Technische Universität München, 1997.

[Spi94] K. Spies. *Funktionale Spezifikation eines Kommunikationsprotokolls.* SFB-Bericht 342/08/94 A, Technische Universität München, 1994.

[Spi95] P.P. Spies. *Vorlesung: Betriebssysteme*, 1995. Vorlesungsskript.

[Spi97a] P.P. Spies. *Vorlesung: Einführung in die Informatik IV – Kapitel 5: Betriebssysteme*, 1997. Vorlesungsskript.

[Spi97b] P.P. Spies. *Vorlesung: Zentrale und verteilte Betriebssysteme*, 1997. Vorlesungsskript.

[Spi98] P.P. Spies. *Betriebssysteme: Modellierung und Realisierungen.* Oldenbourg Verlag, 1998. In Vorbereitung.

[SS95] B. Schätz und K. Spies. *Formale Syntax zur logischen Kernsprache der* FOCUS-*Entwicklungsmethodik.* SFB-Bericht 342/16/95 A, Technische Universität München, 1995.

[Sta91] M.G. Staskauskas. *An Exercise in Verifying Concurrent Programs in Industry: The I/O Subsystem.* in *Proc. Tenth Intl. Phoenix Conf. on Computers and Communications*, 1991, S. 325–331.

[Sta92] W. Stallings. *Operating Systems.* Maxwell Macmillan International Editions, 1992.

[Sto86] W. Stoye. *Message-based Functional Operating System.* Science of Computer Programming **6** (1986), 291–311.

[Stø96] K. Stølen. *Using Relations on Streams to Solve the RPC-Memory Specification Problem.* in *[BMS96a]*, Lecture Notes of Computer Science Bd. 1169. Springer 1996, S. 447–520.

[Tan90] A.S. Tanenbaum. *Betriebssysteme – Entwurf und Realisierung; Teil 1.* Hanser, 1990.

[Tan92] A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[Tan94] A.S. Tanenbaum. *Moderne Betriebssysteme.* Hanser, 1994.

[TB93] C.J. Theaker und G.R. Bookes. *Concepts of Operating Systems.* MacMillan Press, 1993.

[Tho89] B. Thomsen. *A Calculus of Higher Order Communicating Systems.* in *Proc. POPL'89*, Austin, Texas 1989, S. 143–154.

[Web92] R. Weber. *Eine Methodik für die formale Anforderungsspezifikation verteilter Systeme.* SFB-Bericht 342/14/92 A, Technische Universität München, 1992.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A	Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids
342/02/95 A	Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law
342/03/95 A	Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space
342/04/95 A	Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes
342/05/95 A	Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems
342/06/95 A	Maximilian Fuchs: Formal Design of a Model-N Counter
342/07/95 A	Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology
342/08/95 A	Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies
342/09/95 A	Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness
342/10/95 A	Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design
342/11/95 A	Thomas Schnekenburger: The ALDY Load Distribution System
342/12/95 A	Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm
342/13/95 A	Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming
342/14/95 A	Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks
342/15/95 A	Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs
342/16/95 A	Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik

Reihe A

342/17/95 A	Georg Stellner: Using CoCheck on a Network of Workstations
342/18/95 A	Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications
342/19/95 A	Thomas Schnekenburger: Integration of Load Distribution into ParMod-C
342/20/95 A	Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication
342/21/95 A	Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control
342/22/95 A	Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes
342/23/95 A	Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation
342/24/95 A	M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems
342/01/96 A	Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries
342/02/96 A	Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs
342/03/96 A	Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes
342/04/96 A	Thomas Huckle: Efficient Computation of Sparse Approximate Inverses
342/05/96 A	Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification
342/06/96 A	Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components
342/07/96 A	Richard Mayr: Some Results on Basic Parallel Processes
342/08/96 A	Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht
342/09/96 A	P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme
342/10/96 A	Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFSLib – A File System for Parallel Programming Environments
342/11/96 A	Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions
342/12/96 A	Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete
342/13/96 A	Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks

Reihe A

342/14/96 A	Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project
342/15/96 A	Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus
342/16/96 A	Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen
342/17/96 A	Richard Mayr: Model Checking PA-Processes
342/18/96 A	Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States
342/01/97 A	Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsfähigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken
342/02/97 A	Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations
342/03/97 A	Bernhard Mitschang: Technologie für Parallele Datenbanken - Bericht zum Workshop
342/04/97 A	nicht erschienen
342/05/97 A	Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik
342/06/97 A	Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump
342/07/97 A	Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming
342/08/97 A	Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement
342/09/97 A	Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic
342/10/97 A	Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus
342/11/97 A	Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity
342/12/97 A	Christian B. Czech: Architektur und Konzept des Dycos-Kerns
342/13/97 A	Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow
342/14/97 A	Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level
342/15/97 A	Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen
342/16/97 A	Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung

Reihe A

342/17/97 A Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing
342/18/97 A Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations
342/19/97 A Frank Wallner: Model Checking LTL Using Net Unfoldings
342/20/97 A Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation – Fallstudie –
342/21/97 A Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving
342/22/97 A T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0)
342/23/97 A Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP
342/24/97 A Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II
342/25/97 A Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS
342/26/97 A Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration
342/27/97 A Max Breitling: Formalizing and Verifying TimeWarp with FOCUS
342/28/97 A Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase Frame-Work for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation)
342/29/97 A Radu Grosu, Ketil Stølen: Compositional Specification of Mobile Systems
342/01/98 A A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): „Anwendungsbezogene Lastverteilung“, ALV'98
342/02/98 A Ursula Hinkel: Home Shopping - Die Spezifikation einer Kommunikationsanwendung in Focus
342/03/98 A Katharina Spies: Eine Methode zur formalen Modellierung von Betriebssystemkonzepten

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech1: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox -Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support
	Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS